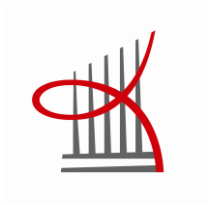


Kaivusyvyysjärjestelmän ohjelmistokehitys

Jarno Mäkipää

Opinnäytetyö
Toukokuu 2012
Tietotekniikka
Ohjelmistotekniikka

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Ohjelmistotekniikan suuntautuminen

MÄKIPÄÄ JARNO:
Kaivusyvyysjärjestelmän ohjelmistokehitys

Opinnäytetyö 47 sivua, josta liitteitä 3 sivua
Toukokuu 2012

Tässä opinnäytetyössä tutkitaan oliopohjaisen ohjelmoinnin etuja sekä haittoja sulautetun reaaliaikajärjestelmän ohjelmistokehityksessä. Tavoitteena on tutkia ja kehittää sulautetussa järjestelmässä toimiva kaivusyvyysjärjestelmän käyttöliittymä, joka on laiteympäristöstään huolimatta mahdollisimman modulaarinen sekä uudelleenkäytettävä.

Kohdealustan tuntemus on sulautetussa ohjelmistokehityksessä hyvin tärkeää. Tästä syystä tässä työssä perehdytään myös kohdeprosessorin ja reaaliaikakäyttöjärjestelmän tarjoamiin ominaisuuksiin.

Kaivusyvyysjärjestelmän käyttöliittymän ominaisuuksien määrittelyssä käytettiin hyödyksi yrityksen aikaisempien tuotteiden hyväksi havaittuja ominaisuuksia sekä asiakkailta saatuja kokemuksia.

Työ on toteutettu Novatron Oy -nimiselle yritykselle, jonka päätoimiala on koneohjausjärjestelmät. Ohjelmistosuunnittelu aloitettiin kevään 2011 aikana ja tuotteen ensimmäinen versio saatettiin markkinoille loppusyksystä 2011, jonka jälkeen tuotetta on jatkokehitetty lisäominaisuuksien muodossa.

Asiasanat: arm, freertos, koneohjausjärjestelmä, olio-ohjelmointi, reaaliaikajärjestelmä

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Software Engineering

MÄKIPÄÄ JARNO:
Software development of excavator grade control system

Bachelor's thesis 47 pages, appendices 3 pages
May 2012

This thesis examines advantages and disadvantages of object-oriented programming in real-time embedded software development. Objective is to develop a user interface to excavator grade control system that works in embedded software environment. And to produce as modular and reusable software as possible in platform that has limited resources.

Knowledge of the target platform is essential in embedded software development. For this reason, this thesis also focuses on investigation of the target processor and the real-time operating characteristics.

Knowledge derived from customer's feedback and successful solutions of earlier products is been used in specification and development of grade control systems user interface.

Development has been done under supervision of Novatron Oy. Main area of business of Novatron Oy is machine control systems. Software development was started in spring of 2011 and the first version of the product was released in late autumn 2011. After first product release development has been continued in form of additional features.

Key words: arm, freertos, machine control system, object-oriented programming, real-time system

SISÄLLYS

ERITYISSANASTO

ARM	*Advanced RISC Machines;* 32-bittinen mikroprosessoriarkkitehtuuri. Yleinen kannettavissa laitteissa ja sulautetuissa järjestelmissä.
DMIPS	Suoritusnopeuden vertailuun käytettävä yksikkö. Yksikkö ilmaisee, kuinka monta miljoonaa kertaa suoritin pystyy suorittamaan Dhrystone-testi silmukan yhdessä sekunnissa.
FLASH	Puolijohdemuisti, joka säilyttää tiedon virran katkaisun jälkeen, ja jota voidaan kirjoittaa sekä lukea sähköisesti.
FPU	*Floating-point unit;* Laitteistototeutettu liukuluku laskin.
FreeRTOS	Reaaliaika käyttöjärjestelmä joka on suunnattu pienille sulautetuille järjestelmille.
IDE	*Integrated Development Enviroment*; integroitu joukko ohjelmointityökaluja
Kellojakso	Prosessorin kellosignaalin yksi jakso. Kaikki suorittimen operaatiot on tahdistettu kellojaksoihin.
Käskykanta	Suorittimen konekieliset käskyt joiden avulla prosessoria ohjataan.
Käyttömuisti	Väliaikaismuisti johon tallentuu ajonaikaiset tiedot.
LSB	*Least Significant Bit;* Vähiten merkitsevä bitti. Bitti jonka muutos vaikuttaa lukuun vähiten.
MSB	*Most Significant Bit;* Eniten merkitsevä bitti. Bitti jonka muutos vaikuttaa lukuun eniten.
MMU	*Memory Management Unit;* Muistinhallinta yksikkö.
Muteksi	Rinnakkaisohjelmoinnissa käytetty resurssin lukitusmekanismi
RISC	*Reduced Instruction Set Computer;* Suoritinarkkitehtuuri jossa konekielen käskyt ovat pidetty yksinkertaisina ja nopeasti vakioajassa suoritettavina.
RAM	*Random Access Memory;* Muistityyppi joka mahdollistaa tiedon käsittelyn satunnaisesta kohdasta vakioajassa. Käytetään yleensä myös kattoterminä termille käyttömuisti.

RTTI	Run-time type information; ajonaikainen olion tyyppi-informaatio
Semafori	Rinnakkaisohjelmoinnissa käytetty lukitusmekanismi
SRAM	*Static Random Access Memory;* Nopea ja vähän virtaa vievä puolijohde muisti. Käytetään usein käyttömuistina.
Takaisinkutsu funktio	*Callback function*; Funktio joka annetaan rajapinnalle kutsuttavaksi, jota rajapinta kutsuu tietyn tapahtuman tapahtuessa. Mahdollistaa keinon alemman tason ohjelmistorajapinnalle kutsua korkeamman tason rajapinnan rutiineja.
Taski	FreeRTOS käyttöjärjestelmässä yksi itsenäinen suoritettava tehtävä.
Thumb	ARM prosessorien käyttötila jossa ohjauskäskyt ovat tavallisuudesta poiketen 16-bitin mittaisia.
Thumb-2	ARM prosessorien käyttötila jossa ohjauskäskyt ovat vaihtuvapituuksisia. Sisältäen vanhan Thumb käskykannan sekä normaalin 32bittisen ARM käskykannan.
UTF-8	UCS Transformation Format; muuttuva pituuksinen merkistö enkoodaus, jolla pystytään esittämään kaikki Unicode-merkistön sisältämät merkit.

1 JOHDANTO

Tässä opinnäytetyössä tutkin oliopohjaisen lähestymistavan etuja sekä rajoitteita sulautetun järjestelmän ohjelmistokehityksessä. Lisäksi käsittelen laitteiston asettamia rajoitteita ohjelmistokehityksen näkökulmasta. Opinnäytetyötä tehdessä keskityn kaivusyvyysjärjestelmän ohjelmistokehitykseen. Kaivusyvyysjärjestelmää toteuttaessani pyrin luomaan toimivan ja helposti hallittavan kokonaisuuden käyttäen olio-pohjaista lähestymistapaa sekä reaaliaikakäyttöjärjestelmän tarjoamia ominaisuuksia.

Olio-ohjelmointi on ollut ohjelmistokehityksen vallitseva trendi pöytätietokone ympäristössä jo vuosikymmenien ajan, mutta sitä on vältetty sulautetuissa järjestelmissä sen resurssivaatimusten takia. Nykyään sulautettujen järjestelmien ohjelmistot saattavat olla kuitenkin suuria ja vaikeasti hallittavia kokonaisuuksia ja laitteiden suorituskyky on kasvanut. Tämä kehityksen muutos antaa sijaa myös olio-ohjelmoinnille perinteisten ohjelmointitekniikoiden rinnalle.

Teoriaosuudessa käsittelen lyhyesti kohdealustan ja reaaliaikakäyttöjärjestelmän rakennetta ja ominaisuuksia, jonka jälkeen paneudun olio-ohjelmoinnin etuihin ja haittoihin. Lisäksi teoriaosuudessa käsittelen kaivusyvyysjärjestelmässä tärkeänä osana olevien matemaattisten operaatioden toimintaan. Työssä en kuitenkaan keskity syvällisemmin siihen kuinka koneohjausjärjestelmän varsinainen laskenta toimii, aihepiirin luottamuksellisuuden vuoksi.

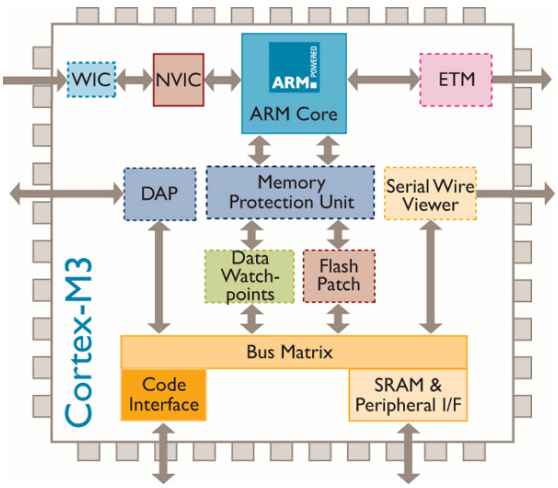
Työn lopussa esittelen kaivusyvyysjärjestelmän käyttöliittymän olio-pohjaisella lähestymistavalla olevaa toteutusta ja lisäksi pohdin saatuja lopputuloksia.

2 Kohdealusta

2.1 STM32 Mikrokontrolleri

STMicroelectronics:in valmistama ARM Cortex-M3 on nykyaikainen Harvard arkkitehtuuriin perustuva 32bittinen suoritin, joka on tarkoitettu laitteille, jotka tarvitsevat hyvää suorituskykyä pienellä virrankulutuksella sekä mahdollisimman pienillä valmistuskustannuksilla. STMicroelectronics valmistaa kyseisestä prosessorista neljää eri versiota, jotka on luokiteltu suorituskyky vaatimuksien mukaan. Suurimmat erot mallien välillä ovat SRAM-muistin ja Flash muistin määrä, sekä liitetyt oheislaitteet. (S. Sadasivan, 2006, 1-17)

Kohdelaitteelta vaaditaan kohtalaista suorituskykyä sekä laajoja käyttäjäominaisuuksia. Tästä syystä kaivusyvyysmittarin prosessoriksi valittiin kyseinen STMicroelectronics:in valmistama ARM Cortex-M3 tyyppinen mikrokontrolleri. Prosessorissa on 512KB Flash ohjelmamuistia sekä 64KB SRAM käyttömuistia. Prosessorin kellotaajuus on 72MHz ja se pystyy 1.25DMIPS/MHz laskentanopeuteen. Kaivusyvyysjärjestelmän on toimittava vaihtelevissa lämpötilaolosuhteissa. Laitteen toimivuuden vaihtelevissa olosuhteissa takaa mikro-ohjaimen -40°C ~ 85°C sallittu käyttölämpötila alue. Mikro-ohjaimen tyyppinumero STM32F103RET6. (STMicroelectronics, 2009, STM32F103xx datasheet, 1, 40) Kuvassa 1 nähdään lohkokaavio prosessorin sisäisestä rakenteesta.

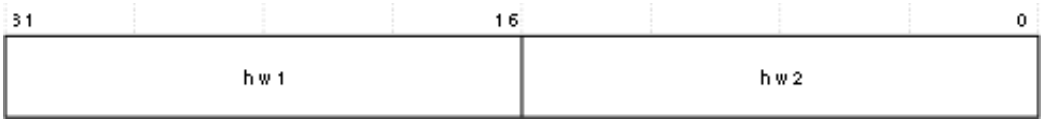


KUVA 1. ARM Cortex-M3 sisäinen rakenne
(http://www.embeddedinsights.com/epd/arm/arm-cortex-m3.php, viitattu 25.3.2012)

2.1.1 Käskykannat

Cortex-M3 prosessori sisältää uudentyyppisen Thumb-2 käskykannan. Se on vaihtuvapituuksinen käskykanta, jonka tavoitteena on mahdollistaa mahdollisimman hyvä suoritustuskoodin tiiviys säilyttäen kuitenkin 32-bittisen prosessorin käskykannan tehokkuus. 16-bittiset käskyt vievät vähemmän ohjelmamuistia sekä kuormittavat vähemmän prosessorin ja ohjelmamuistin välistä muistiväylää. Thumb-2 sisältää joukon 16-bittisiä Thumb-käskyjä laajennettuna 32-bittisillä ARM käskykannan käskyillä. ARM:in tekemät tutkimukset osoittavat, että Thumb-2 teknologia käyttää 31 prosenttia vähemmän muistia kuin perinteinen 32-bittinen ARM käskykanta ja tuottaa 38 prosenttia tehokkaampaa suoritusta, kuin pelkkä 16-bittinen Thumb-käskykanta. Se onkin oikeastaan kompromissi 16-ja 32-bittisten käskykantojen välillä. (ARM, 2012, The ARM Processor Architecture)

Kuvassa 2 on esitetty Thumb-2 ohjauskäskyn rakenne. Ensimmäiset 16-bittiä *hw1* määrittää ohjauskäskyn pituuden, sekä toiminnallisuuden. Jos prosessori tulkitsee käskyn 32-bittiseksi, se hakee myös seuraavat 16bittiä *hw2*. Suurimpana lisäyksenä 32-bittiset käskyt antavat 16-bitin Thumb-käskyihin ovat tuki keskeytyskäsittelylle, apuprosessorien käsittely sekä digitaaliseen signaalin käsittelyyn liittyvät käskyt. (ARM, 2007, Cortex-A8 Technical Reference Manual, 50)



KUVA 2. Thumb-2 ohjauskäskyn rakenne

2.1.2 Laskentateho

Prosessorin eri ohjauskäskyjen suoritusnopeutta voidaan mitata CPI arvon perusteella (Cycles Per instruction) eli kellojaksojen määrä per ohjauskäsky. Näitä tutkimalla saadaan selville kuinka kauan prosessorin erityyppisten ohjauskäskyjen suorittaminen kestää.

ARM Cortex-M3 sisältää aritmeettis-loogisen yksikön, jossa on tuki mm. 1 kellojakson kertolasku operaatiolle, 2-14 kellojakson jakolaskulle riippuen arvojen koosta (ARM,

2007, Cortex-M3 Technical Reference manual, s 18-3). Nämä koskevat kuitenkin vain maksimissaan 32 bittisiä kokonaislukuja, eikä laitteessa ole laitteistopohjaista liukuluku yksikköä eli FPU:ta, joten kaikki liukulukulaskenta suoritetaan ohjelmallisesti. Sourcery G++ Lite tuottamat ohjelmalliset liukuluku laskut ovat kertolaskun osalta 23-36 kellojaksoa sekä jakolasku 23-152 kellojaksoa. (P.Aimonen, 2012, Libfixmath benchmarks.) Vertailun vuoksi voidaan kertoa, että esimerkiksi harrastekäyttöönkin hyvin yleisessä mikrokontrollerissa AVR Atmega128 ei ole laitteistopohjaista tukea jakolasku operaatiolle ollenkaan ja kertolaskukin vie 2 kellojaksoa (Atmel, Atmega128/L Datasheet Summary, s 9-11). Sellaisissa tilanteissa, joissa ei ole jakolaskuille laitteistototeutusta, selvitään usein korvaamalla jakolaskut bittisiirroilla. C-kääntäjästä toki löytyy myös ohjelmallinen toteutus jakolaskulle, mutta sen tehokkuudesta ei ole takeita. On myös olemassa useita optimoituja assembly toteutuksia AVR mikro-ohjaimelle jakolaskusta, mutta niiden tehokkuus jää 600 kellojaksoon. Joten tästä voisi päätellä, että ARM prosessorissa riittää laskentatehoa ainakin teoreettisella tasolla.

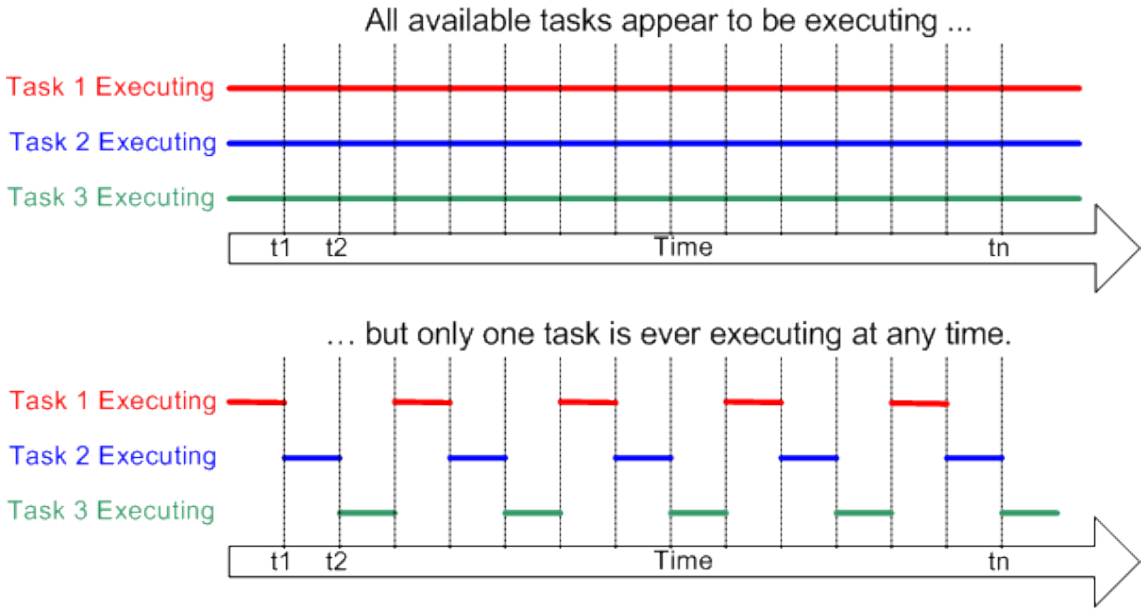
2.2 FreeRTOS

FreeRTOS on reaaliaika käyttöjärjestelmä, joka on suunnattu pienille sulautetuille järjestelmille. Se tukee yli 30 eri prosessoriarkkitehtuuria, joihin myös kohdelaite ARM Cortex M3 lukeutuu. FreeRTOS antaa tuen rinnakkaisille prosesseille, jota FreeRTOS dokumentaatioissa kutsutaan taskeiksi. Lisäksi käyttöjärjestelmä tarjoaa myös geneeriset jonot, semaforit ja muteksit prosessien väliseen kommunikoitiin toisiensa välillä sekä keskeytysten kanssa. (R.Barry, 2010, Features Overview.)

Reaaliaikakäyttöjärjestelmä tarkoittaa periaatteen tasolla sitä, että se suorittaa sille annettuja tehtäviä reaaliaikaisesti. Tämä ei ole kuitenkaan yleisesti ole täysin mahdollista edes teoreettisella tasolla, mutta voidaan ehkä paremmin kuvata, että reaaliaikajärjestelmän tarkoitus on taata ajoituksen ennustettavuus. Reaaliaikajärjestelmät voidaan jakaa koviin ja pehmeisiin järjestelmiin. Pehmeässä reaaliaikajärjestelmässä prosesseille määritellään aikaikkuna, jossa niiden kuuluisi suoriutua. Jos suoritus venähtää ei siitä nosteta virhettä vaan on parempi, että suoritetaan toiminto loppuun. Kova reaaliaikajärjestelmä sen sijaan lopettaa kyseisen prosessin suorituksen, silloin kun ei kykene suoriutumaan sille määritellyssä ajassa. FreeRTOS toteuttaa pehmeän reaaliaikakäyttöjärjestelmän vaatimukset, sillä siinä ei ole mahdollisuutta asettaa prosesseille suoritusaika rajoitteita. (TTY, Sulautettu ohjelmointi, Skedulointi eli vuoronnus, 100-102.)

2.2.1 Rinnakkaiset prosessit ja vuorottaja

Normaali yksiytiminen prosessori ei normaalitilanteessa pysty suorittamaan kuin yhtä tehtävää kerrallaan. Silti usein käyttöjärjestelmä saa tilanteen näyttämään siltä, kuin useaa tehtävää suoritettaisiin samanaikaisesti. Käyttöjärjestelmä hoitaa tämän asian pätkimällä prosessien suoritusta vuorottaen prosessoriaikaa kullekin prosessille (R.Barry,2010, RTOS Fundamentals). Kuvasta 2 voimme nähdä miltä taskien vuorotus voi periaatetasolla näyttää.

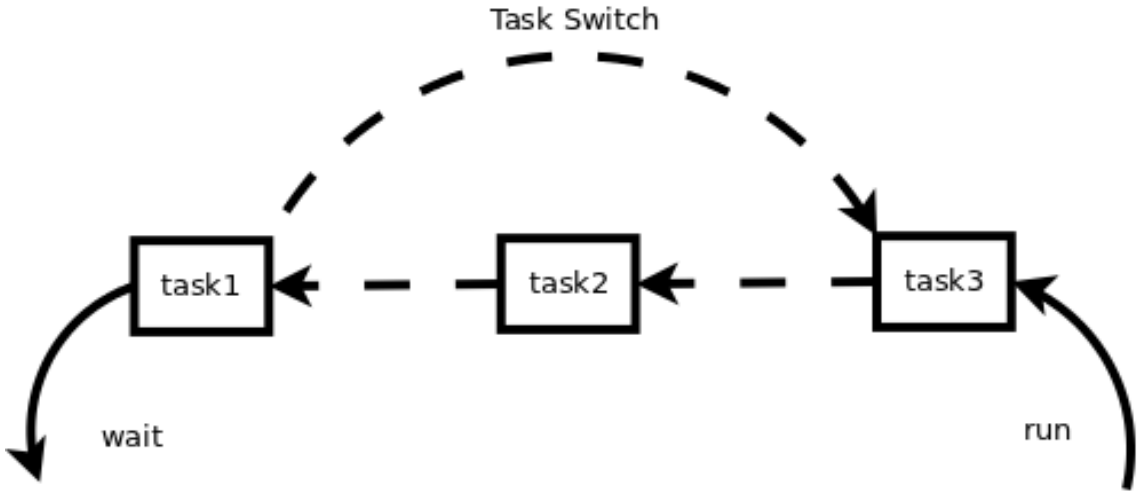


KUVA 2. Taskien suoritus
(http://www.freertos.org/implementation/a00004.html,viitattu 25.3.2012)

FreeRTOS käyttää kiinteän prioriteetin vuorottelumenetelmää (fixed priority skeduling). Tämä eroaa normaalista kiertovuorottelusta (round robin) siten, että korkeamman prioriteetin prosessi saa suoritusaikaa ensin. Perusperiaate on se, että prosessia suoritetaan niin kauan kunnes on kulunut yksi aikamääre, ellei sen suoritus ole ennen sitä päättynyt. Sen jälkeen vuorottaja tarkistaa onko samalla tai korkeammalla prioriteetilla olevia prosesseja ja siirtyy suorittamaan yhtä niistä seuraavan aikamääreen ajaksi.

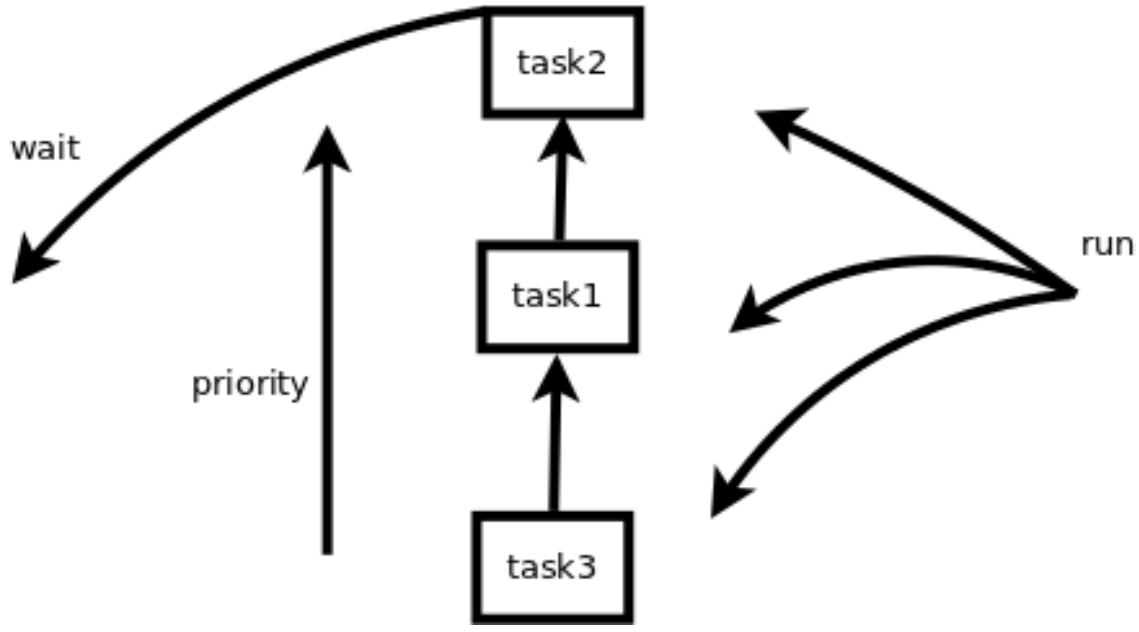
Kuvassa 3 näemme perinteisen kiertovuorottelun periaatteen. *Task1* on asetettu suoritusjonoon ensimmäiseksi, mutta sitä ei ehditä suorittaa loppuun ennen aikaviipaleen päättymistä. Tällöin vuorottaja tekee taskin vaihdoksen ja antaa suoritusaikaa suoritus-

jonon viimeiselle taskille, josta se hyppää seuraavaan kun aikaviipale on päättynyt tai taski on suoritettu loppuun.



KUVA 3. Taskien kiertovuoroittelu ilman prioriteettiä

Kuvassa 4 näemme karkealla tasolla kuinka kiertovuoroittelu prioriteetin kanssa toimii. Taskin tullessa suoritustilaan, se asetetaan vuorottajan suoritusjonoon prioriteetin mukaisesti. Vuorottaja antaa suoritusaikaa aina korkeimmalla prioriteetilla olevalle säikeelle, ja jos suoritettavana on kaksi tai useampi saman prioriteetin omaavia säikeitä, vuorottaja jakaa kiertovuoroittelun periaatteiden mukaisesti näille säikeille suoritusresursseja aikaikkuna kerrallaan.



KUVA 4. Taskien kiertovuoroittelu prioriteetin kanssa

Kiinteän prioriteetin vuoroittelu sopii erittäin hyvin reaaliaikajärjestelmiin. Se nimittäin takaa, että korkeimman prioriteetin säikeet tulevat aina suoritetuiksi ensiksi, silloin kun ne ovat valmiita suoritettaviksi. Tämä vuoronnusmenetelmä on hyvä etenkin silloin, kun tehdään järjestelmää, jossa on samanaikaisesti käsiteltävänä jotain vähemmän tärkeää kuten käyttöliittymä ja reaaliaikaisuutta vaativaa kuten esimerkiksi tietoliikenteen välitystä (TTY, Sulautettu ohjelmointi, Skedulointi eli vuoronnus, 100-102)

2.2.2 Synkronointi

FreeRTOS sisältää useita keinoja kommunikoida prosessien välillä. Näitä ovat jonot, mutexit sekä semaforit.

Jonot

Jonot mahdollistavat keinon muodostaa yhteyksiä taskien välillä tai taskin ja keskeytyksen välillä. FreeRTOS jono toimii parhaiten normaalina säie turvallisena FIFO puskurina, esimerkiksi siten, että yhdessä taskissa kirjoitetaan jonon perään viestejä ja toisessa taskissa luetaan viestejä jonon alusta. Normaalista FIFO käyttäytymisestä poiketen kyseiseen jonoon on mahdollista kirjoittaa myös alkuun. Jonoviestien kuuluu olla jonon sisällä keskenään vakiokokoiset ja jonoa luotaessa on määriteltävä jonon maksimikoko. Jonoviestien koko kannattaa pitää mahdollisimman pienenä, sillä viestit kopioidaan jonoon kokonaisuudessaan. (R.Barry,2010, Inter-task Communication)

Semaforit

Binäärisemaforeja voidaan käyttää sekä jaettujen muuttujien suojaamiseen että synkronointiin. Muuttujien suojaamistarkoituksessa ne toimivat hyvin samalla tavalla kuin mutexit, mutta ne eivät sisällä prioriteetin perintä ominaisuutta. Suojattavan asian ei tietenkään tarvitse olla muuttuja vaan se voi olla esimerkiksi liitetty oheislaite, esimerkiksi I2C väylässä oleva EEPROM muistipiiri, johon tallennetaan asetuksia tai sarjaliikenneportti, jolla lähetetään tietoa toisille laitteille. Semaforien käyttäminen on usein hyvin helppoa. Ennen jaetun resurssin käyttöä pyydetään semaforia prosessille kutsuen *xSemaphoreTake()* kutsua. Kun resurssin käyttö on lopetettu, annetaan se pois prosessin käytöstä kutsuen *xSemaphoreGive()* kutsua. Semaforin pyyntövaiheessa sitä voidaan jäädä odottamaan aikamääreeksi, loputtomiin tai jatkaa suoraan suoritusta. Jos taski ei

saa resurssia suoraan ja jää odottelemaan sitä, se menee blocked tilaan ja antaa täten resursseja toisille taskeille. (R.Barry,2010, Inter-task Communication)

Semaforeja voidaan käyttää myös prosessien käynnistämiseen. Tämä on erittäin kätevää kun halutaan esimerkiksi pyörittää jotain taskia vain silloin kun joku keskeytys tai muu tapahtuma on tapahtunut. Seuraavana näemme listauksen, jossa esitellään tapa käyttää semaforeja taskin synkronointiin.

```
xSemaphoreHandle xSemaphore = NULL;

void vATask( void * pvParameters )
{
    vSemaphoreCreateBinary( xSemaphore );

    /*task will loop here forever*/
    for( ;; ) {

        if( xSemaphoreTake( xSemaphore, LONG_TIME ) == pdTRUE ) {

                ... do actions
        }
    }
}

void vISR( void * pvParameters )
{
    ... Do functions.

   /* Unblock the task by releasing the semaphore. */
   xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );
   portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

LISTAUS 1. Taskin synkronointi käyttäen semaforia.

Listauksessa 1 näkyy taskin liitos funktio *vATask* ja keskeytysfunktion *vISR*, joka antaa taskille suoritusluvan. Ohjelmistolistauksesta nähdään kuinka taski luo semaforin ja menee tämän jälkeen ikuiseen silmukkaan odottelemaan, että semafori vapautuu ja suorittaa sen vapauduttua sille annettuja tehtäviä. Semaforia ei kuitenkaan vapauteta tehtävien jälkeen taskin sisällä, vaan taski jää taas silmukan uudella kierroksella odottamaan samaa semaforia, jonka se oli juuri itse varannut. Tämä ei ole ohjelmistovirhe vaan haluttu ominaisuus. Semaforin nimittäin vapauttaa alempana näkyvä keskeytysfunktio *vISR* keskeytyksen tapahduttua vapauttaa semaforin ja tällöin taski saa taas suoritusoikeuden. Tällaista lähestymistapaa voidaan käyttää esimerkiksi silloin, kun joltain ulkoiselta laitteelta aletaan vastaanottaa informaatiota ja tämä aiheuttaa alkutilanteessa keskeytyksen. Prosessoriaikaa paljon vievät asiat kannattaa hoitaa taskien sisällä eikä kes-

keytyksissä, koska keskeytykset suoritetaan aina korkeammalla prioriteetillä kuin taskit. Tästä syystä monessa tilanteessa tulee keskeytyksen sisällä hoitaa vain välttämättömimmät asiat ja käynnistää prosessointi taski, jottei prosessointi häiritse järjestelmän muuta toimintaa. (R.Barry,2010, Inter-task Communication)

Mutexit

Mutekseja voidaan käyttää semaforien tapaan jaettujen resurssien suojaamiseen. Näiden kahden välillä on suurinpana erona muteksien kyky muuttaa suoritettavan taskin prioriteettiä hetkellisesti. Tämä tapahtuu seuraavasti: kun matalammalla prioriteetillä oleva taski ottaa haltuunsa muteksin ja aloittaa suorittamaan tehtäviään, mutta sen suoritus jää kuitenkin kesken, kun korkeamman prioriteetin taski aloittaa suorituksen. Korkeamman prioriteetin taski päätyy kuitenkin tilanteeseen, että sen tarvitsee käyttää samaa resurssia ja yrittää ottaa itselleen saman muteksin. Tällöin korkeamman prioriteetin taskin jäädessä odottamaan, se antaa oman prioriteettinsa hetkellisesti muteksin haltijalle ja tämä pääsee suorittamaan tehtävänsä siihen vaiheeseen, että voi taas vapauttaa muteksin korkeamman prioriteetin taskille.

Mutekseista on olemassa myös rekursiiviset versiot, joita voi nimensä mukaisesti kutsua rekursiivisten funktioiden sisällä. Nämä pitävät sisällään laskurin, joka kasvattaa arvoaan yhdellä aina kun sama taski varaa sitä ja laskee yhdellä kun rekursiivista muteksia vapautetaan. Jolloin muteksi on vapaa toisille taskeille vasta kun sen laskuri on täysin tyhjä. Rekursiiviset muteksit ovat käteviä kun niitä käytetään oikein, mutta altistavat usein mahdollisille muteksien kuolonlukoille.

Mutekseja ei voi käyttää keskeytysten sisältä koska ne pitävät sisällään informaation niiden varanneesta taskista. Näitä tapauksia varten voi kuitenkin käyttää binäärisemaforia ainakin useissa tapauksissa, ja muissa tapauksissa kannattaa miettiä lähestymistapa erilaiseksi.

2.2.3 Ajastimet

FreeRTOS antaa tuen ohjelmallisille ajastimille. Ajastimet mahdollistavat funktioiden suorituksen aikamääreen jälkeen tai tietyn intervallin välein. Funktio jota ajastin kutsuu aikamääreen tulessa umpeen, kutsutaan ajastimen takaisinkutsufunktioksi.

Ajastimet on toteutettu FreeRTOS:issa siten, että taustalla pyörii ajastin palvelu, eli niin sanottu demoni. Kun luodaan uusi ajastin, se menee demonin ajastinjonoon ja demonitaski suorittaa sen kun sen aikamääre on tullut umpeen. Ajastinfunktiota toteuttaessa pitää ottaa huomioon, että takaisinkutsufunktiot kutsutaan ajastin taskin sisältä, joten ne eivät saa olla blokkaavia. Takaisinkutsussa ei siis kannata pyytää ja jäädä odottamaan esimerkiksi muteksia tai semaforia. Sillä jos takaisinkutsufunktio on blokkaava, se häiritsee ajastin taskin toimintaa, ja estää muiden ajastinfunktioiden suorittamisen. (R.Barry,2010, Software Timers)

FreeRTOS ajastimet tarjoavat hyvän ratkaisun kun halutaan suorittaa tehtäviä aikaintervallilla joka poikkeaa taskien normaalista suoritus intervallista.

2.3 CodeSourcery G++ Lite ARM-kehitysympäristö

Sourcery G++ lite kääntäjä on suunnattu sulautettujen ARM pohjaisten ohjelmistojen kehitykseen. Etenkin silloin kun kehitetään ohjelmistoa suoraan laitteiston päälle ilman käyttöjärjestelmää tai kun käytetään kevyttä reaaliaikakäyttöjärjestelmää. Se ei kuitenkaan sovellu esimerkiksi isompien käyttöjärjestelmien, kuten Linux tai uClinux ytimen kääntämiseen tai ohjelmistokehitykseen. (CodeSourcery, 2010, 2)

Sourceryn G++ lite kehitysympäristö on GNU kehitystyökalu ympäristön päälle rakennettu lisäosa, jonka kohdealue on erityyppiset ARM suorittimet. Kääntöympäristö on GNU GPL lisenssin alla, joten sen lähdekoodit ovat vapaassa levityksessä kaikille. Ilmaiseen versioon kuuluu kuitenkin vain komentorivityökalut. Kehitysympäristöstä on olemassa myös kaupallinen versio, johon sisältyy kääntötyökalujen lisäksi myös graafinen IDE, josta löytyy valmiit asetuskokoonpanot useimmille kehitysalustoille. Kaupallisen version IDE on kuitenkin vain myös vapaalla lisenssillä olevan Eclipse nimisen ohjelmointiympäristön päälle toteutettu ratkaisu.

3 Olio-ohjelmointi sulautetussa järjestelmässä

Embedded C++ Technical Committee julkaisi vuonna 1999 määrittelyn C++ ohjelmointikielen rajoitetusta versiosta jonka kohdealue olisi erityisesti sulautetut järjestelmät. Määrittely oli osajoukko C++ kielestä, josta oli poistettu ominaisuuksia, jotka katsottiin suoranaisesti vaikuttavan kielen tehokkuuteen. C++ kielen yleisimmistä ominaisuuksista oli poistettu: moniperintä, virtuaaliset kantaluokat, ajonaikainen tyyppi-informaatio, nimiavaruudet, poikkeuskäsittely sekä mallit. (Embedded C++ Technical Committee, 1999) Määrittelyssä pyrittiin kuitenkin säilyttämään C++ kielen oliopiirteet.

Embedded C++ määrittely ei kuitenkaan koskaan yleistynyt. Tätä nykyä ei ole kovinkaan montaa kääntäjää joka tukisi sitä. Sen sijaan jotkut kääntäjät tukevat C++ ohjelmointikieltä kaikilla yleisillä ominaisuuksilla, kuten käyttämäni CodeSourcery G++ Lite tukee. Joitakin ominaisuuksia ei voi kuitenkaan käyttää johtuen laitteiston asettamista rajoitteista. Käytettävien ominaisuuksien rajoittaminen jääkin ohjelmistokehittäjän omalle vastuulle.

3.1 C++ ohjelmointikielen edut ja haitat sulautetussa järjestelmässä

Olio-ohjelmointi ei itsessään sisällä mitään sellaista joka vaikuttaisi kriittisesti tehokkuuteen. Kuitenkin jotkut ominaisuudet joita esimerkiksi C++ sisältää tuo mukanaan tehokkuuteen vaikuttavia tekijöitä. Myös C ohjelmointikieli mahdollistaa olio-pohjaisen ohjelmoinnin.

3.1.1 Edut

Tiedon kapselointi ja piilotus

Olio-ohjelmoinnin perusidea on luoda yksittäisiä kokonaisuuksia ja piilottaa niistä ne ominaisuudet joiden ei tarvitse näkyä ulospäin. Tämä on tapa estää mahdolliset väärinkäytöt ja estää sen aiheuttamat virhetilanteet. Lisäksi tiedon piilotuksella mahdollistetaan myös parempi luettavuus sillä rajapinnan käyttäjälle pyritään pitämään esillä vain välttämätön informaatio. Tiedon kapselointi pieniin kokonaisuuksiin puolestaan helpottaa ohjelmistokehityksen hallittavuutta, ylläpitoa sekä mahdollistaa ohjelmakoodin uudelleenkäytettävyyden. (Jan Skansholm, 2002, C++ From the beginning, 211-214)

Periytyminen

Periytyminen on olio-ohjelmoinnin merkittävin ominaisuus, sillä se mahdollistaa geneerisen ohjelmoinnin. Sen perusidea on se, että luodessa uutta komponenttia tutkitaan voitaisiinko käyttää jotain ennalta olevaa komponenttia runkona ja lisätä siihen vain ne ominaisuudet jotka olisivat uudelle komponentille ominaiset. Periytyminen on asia joka oikein toteutettuna jopa vähentää suoritettavan ohjelmakoodin määrää. Tämä johtuu siitä, että jossain muussa toteutusmallissa jouduttaisiin toteuttamaan samoja asioita useampaan kertaan. Kun periyttämällä voidaan uudelleen käyttää samaa ominaisuutta periytetyssä luokassa. (Jan Skansholm, 2002, C++ From the beginning, 211-214)

Yleisesti

Olio-ohjelmointi mahdollistaa ratkaisumallin, joka muistuttaa normaalia ympärillä olevaa maailmaa. Ongelmat voi helposti pilkkoa pieniin helposti hallittaviin osiin. Ja lisäksi ongelman muuttuessa voidaan ottaa valmiista ratkaisusta ne osat, jotka yhtenevät uuden tehtävän kanssa ja toteuttaa vain ne osat, jotka ovat muuttuneet.

3.1.2 Haitat

Muistijalanjälki

C++ tarjoaa paljon laajoja ja kehittyneitä ominaisuuksia. Nämä ominaisuudet vaativat kuitenkin suuria määriä ohjelmamuistia sekä käyttömuistia. Tämä ei ole ongelma normaalissa PC- ympäristössä jossa on muistia useita gigatavuja, mutta järjestelmissä joissa on rajalliset resurssit tämä on usein esteenä. (J.Fisher,P.Faraboschi,C.Young, 2005, Embedded Computing, 447-450)

Determistisyyden puute

Reaaliaika järjestelmän tärkein ominaisuus on sen toiminnan ennustettavuus. Järjestelmälle suunnitelluista tehtävistä pitää suoriutua niille varatussa ajassa tai niiden tapahtuminen pitää taata kaikissa olosuhteissa. Perinteisen C-ohjelmointikielen suoritus rakenne on hyvin yksinkertainen. Jokainen C-kielen rakenne maksaa vakiomäärän järjestelmän resursseja. C-ohjelmakoodista pystytään usein tarkkaan ennustamaan minkälaista konekoodia siitä syntyy. (J.Fisher,P.Faraboschi,C.Young, 2005, Embedded Computing, 445)

C++-kielen standardi sisältää määrittelyn jonka perusteella sen ominaisuudet toimivat, mutta se ei sisällä määrittelyä kuinka ominaisuuden kuuluu sisäisesti olla kääntäjässä toteutettu. Tästä syystä ei ole taattua, että jokin ominaisuus vie tietyn määrän resursseja eri valmistajien C++ kääntäjissä, tai edes saman valmistajan eri versioissa. Sekä ominaisuuden suoritusaika, että muistijalanjälki voi vaihdella. Lisäksi jotkut kielen ominaisuudet rikkovat suoraan reaaliaikaisuuden periaatteita. Esimerkiksi poikkeuskäsittely toimii täysin reaaliaikajärjestelmän vastaisesti. (J.Fisher, P.Faraboschi, C.Young, 2005, Embedded Computing, 447-450)

Lisäksi monet C++ ominaisuudet tukeutuvat dynaamiseen muistinhallintaan. Näistä muutamia ovat esimerkiksi C++ standardi kirjaston tarjoamat *String* sekä *Vector* Luokat. Dynaamista muistinhallintaa ei usein ole sulautetuissa järjestelmissä tarjolla, tai sen käyttäminen muuttaa järjestelmän vasteajan ennustettavuutta.

3.1.3 Asiat joita tulisi välttää

RTTI

Run Time type-information on ajonaikainen tyyppi-informaatio jonka avulla C++:ssa tehdään erityisiä dynaamisia tyyppimuunnoksia. Ominaisuuden päällä pitäminen aiheuttaa kuitenkin noin 40 tavun muistijalanjäljen toteutettavaa luokkaa kohti. Tämä informaatio pitää sisällään muun muassa merkkijonon joka sisältää luokan nimen, muutaman tavun informaatiota luokasta, ja muutaman tavun informaatiota jokaisesta sen perimästä kantaluokasta. (ISO/IEC TR 18015:2006, 2006,Technical Report on C++ Performance,

24) 40 tavua ei itsessään ole paljoa, mutta ominaisuuden antamiin vähäisiin hyötyihin nähden tämä kuluttaa paljon muistia. RTTI:in saa kuitenkin kytkettyä pois useimmista kääntäjistä.

Poikkeukset

Monissa kääntäjissä poikkeuskäsittelyn käyttäminen vaatii myös RTTI:n päälle kytkentää mikä on jo sinällään syy, ettei tätä voida hyödyntää. Tavanomainen C++ poikkeustunniste sisältää 13 osoitinta sekä kaksi 16bittistä kokonaislukua. Jokaisen keskeytyksen aiheuttama jalanjälki on siis noin 56 tavua. CodeSourceryn G++ Lite:n tarjoama toteutus poikkeuksista käyttää lisäksi dynaamista muistinvarausta keskeytysolion luontiin. Tämän salliminen ei ole suotavaa koska laitteessa ei ole dynaamiselle muistinvaraukselle laitetukea. Keskeytysten välttäminen valitettavasti estää monen C++ standardi kirjaston ominaisuuden käyttämisen. (GNU, 2011, GCC lähdekoodi)

Muistijalanjäljen lisäksi poikkeuskäsittely tekee ohjelman vasteajan ennustettavuuden mahdottomaksi. Poikkeuskäsittelijä kutsuu *Try* lohkossa alustettujen olioiden purkajia aina kun poikkeus tapahtuu. Reaaliaikajärjestelmässä tulisi muutenkin toteuttaa prosessit niin ettei poikkeustilanteita synny. (J.Fisher, P.Faraboschi, C.Young, 2005, Embedded Computing, 449)

Mallit

Mallien käyttö saattaa tuottaa olettamattoman suuren määrän ohjelmakoodia. Sillä luokka malli sekä funktio malli tuottaa jokaisella kerralla uuden toteutuksen mallikoodista kun mallia käytetään uudentyyppisillä malliparametreillä. Mallien tuottaman ohjelmakoodin määrä vaihtelee kääntäjäkohtaisesti. (ISO/IEC TR 18015:2006, 2006,Technical Report on C++ Performance, 42-45)

3.2 Muistinhallinta

Kohdealusta ei sisällä laitteistopohjaista muistinhallinta yksikköä (MMU). Näin ollen se ei sisällä myöskään pöytätietokoneista tuttua virtuaalista muistia. Virtuaalisen muistin puuttuminen aiheuttaa ongelmia jos varataan muistia keosta.

3.2.1 Kiinteä muistinvaraus

Kiinteä muistinvaraus tarkoittaa sitä, että objektin muisti varataan ennen ohjelman suorituksen alkamista ja se pysyy ohjelman suorituksen ajan käytettävissä. Ohjelmassa luodut globaalit muuttujat käyttävät siis kiinteää muistinvarausta. Kiinteän muistinvarauksen etuja on ehdoton determistisyys. Mikäli kaikki muisti varataan ohjelman käynnistyessä, ei ohjelman suorituksen aikana synny virheitä muistinvaraukseen liittyen sekä ohjelmistokehittäjällä on aina tieto kuinka paljon sovellus voi maksimissaan käyttää muistia. Haittoja on suuri muistintarve, sillä kaikki ohjelman vaatima muisti pitää olla samanaikaisesti varattuna.

3.2.2 Muistinvaraus pinosta

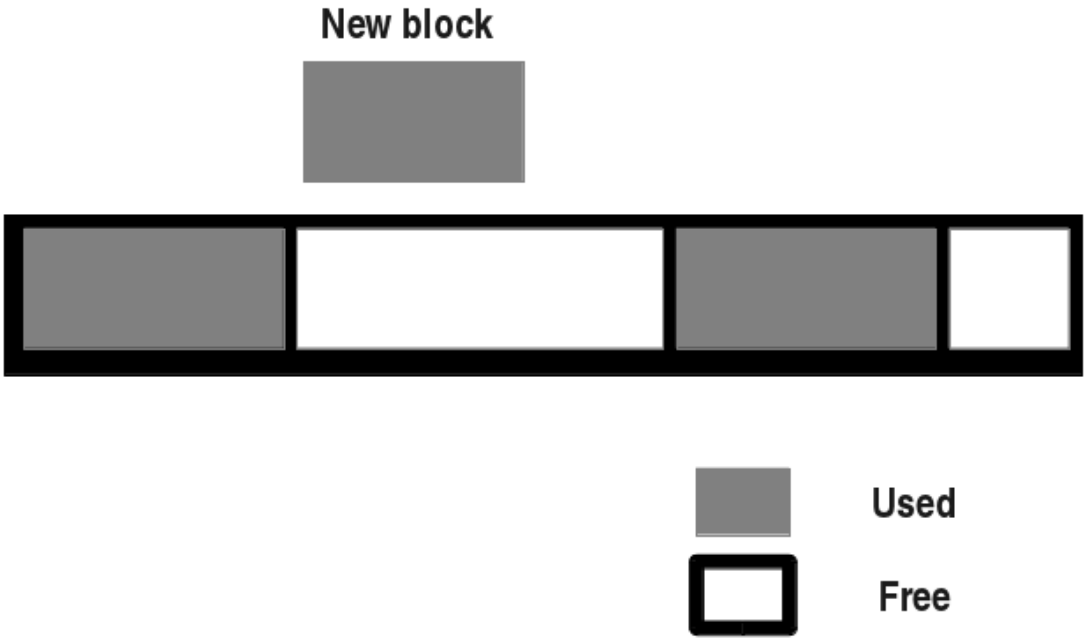
Muistinvaraus pinosta on yleensä itsestään selvää C ja C++ ohjelmoijalle. C/C++ funktion sisällä luodut paikalliset muuttujat varataan aina pinosta ja ne poistuvat pinosta aina kun poistutaan funktiosta. Tämä muistinvaraustyyppi on kaikista nopein, edullisin ja suositeltavin. Siinä on kuitenkin sellainen huono puoli, että pinosta varatuilla objekteilla on vain määrätyn mittainen elinkaari. Etenkin olio-ohjelmoinnissa tulee usein vastaan tilanteita joihin pinoon perustuva muistinvaraus ei sovellu. (C.Weir; J.Noble, 2000, Memory Allocation, 22)

3.2.3 Variable allocation eli muistinvaraus keosta

Lähes kaikki oliopohjaiset kielet perustuvat dynaamiseen muistinvaraukseen eli muistinvaraukseen keosta. Ohjelmointikielissä kuten Java käytetään yksinomaan pelkästään dynaamista muistinvarausta. Tässä muistinvaraustyypissä on etuina se, että varatun objetin voi varata lähes missä tahansa, käyttää sitä vaikkapa jossain toisaalla ja pitää käytössä niin pitkään kuin on tarve. Muistinvaraus keosta kuitenkin aiheuttaa muistin pirs-

taloitumista, mikäli ei ole käytössä laitteistopohjaista muistinhallintayksikköä joka järjestelee varatut muistialueet vierekkäin, yrittäen säilyttää varaamattoman muistialueen mahdollisimman suurena.

Kuvassa 5 on esimerkki dynaamisen muistinvarauksen aiheuttamasta muistin pirstaloitumisesta. Mikäli kuvassa näkyvä muistinalue varataan kyseiseen kohtaan, ei pystytä varaamaan enää toista yhtä suurta muistialuetta, vaikka vapaata tilaa olisikin vielä jäljellä yhteensä riittävästi. Silloin kun on käytössä muistinhallinta yksikkö ja sen tarjoamat virtuaaliset muistiosoitteet, voi muistinhallintayksikkö järjestellä muistialueet järkevämmin. Tämä ei ole mahdollista ilman virtuaalista muistialuetta, koska ohjelman käyttämät muistiosoitteet muuttuisivat.

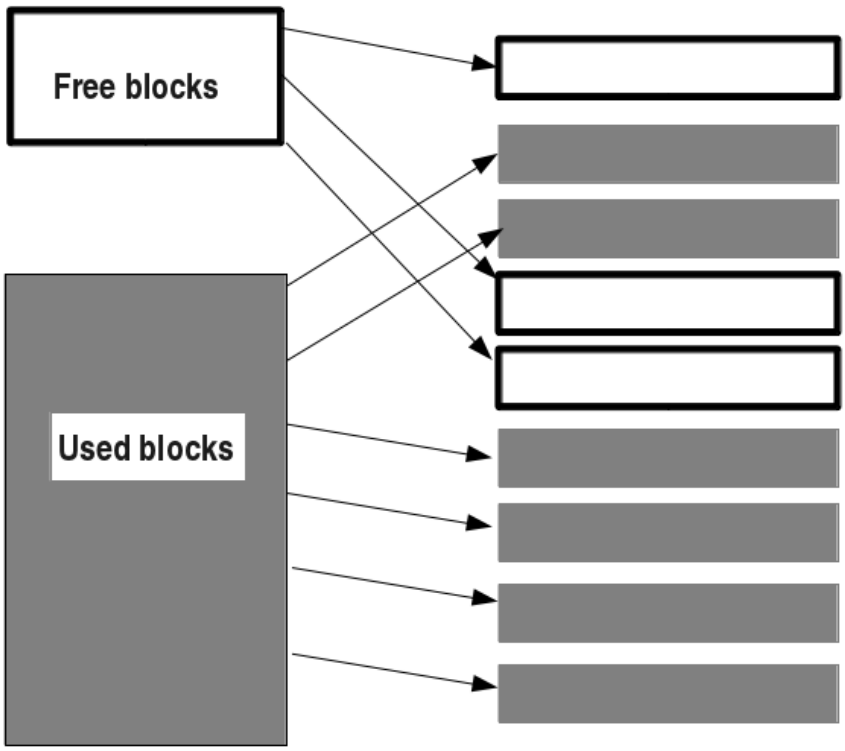


KUVA 5: Muistin pirstaloituminen

3.2.4 Pooled allocation eli muistinvaraus muistialtaasta

Muistiallas tyyppisessä muistinkäsittelyssä varataan järjestelmän suorituksen alussa suuri lohko muistia kiinteällä muistinvarauksella, ja jaotellaan se tietyn kokoisiin lohkoihin. Kun objektille varataan muistia, annetaan sille yksi kokonainen lohko tästä niin sanotusta muistialtaasta. Varattava objekti ei siis voi olla isompi kuin yksittäinen muistialtaan lohko, mutta se voi olla pienempi. Menetelmä soveltuu hyvin tapauksiin joissa

varattavana on paljon samankokoisia pieniä objekteja. Se kuitenkaan sovellu käyttötarkoituksiin joissa on paljon erikokoisia varattavia objekteja koska altaan muistilohkon koon on oltava isoimman varattavan objektin kokoinen, ja kaikki tätä pienemmät varaukset aiheuttavat hukka tilaa. Muistiallas soveltuu kuitenkin useimpiin tilanteisiin jossa olio-ohjelmoinnissa normaalisti on totuttu käyttämään dynaamista muistinhallintaa. Ja on suhteellisen helppo toteuttaa. Kuvassa 6 näkyy muistiallas tyyppisen muistinvarauksen perusidea. Kuvassa oikealla puolella näkyy kuinka muistialtaan lohkot sijoittuvat fyysiseen muistiin, sekä vasemmalla näkyy varatun ja varaamattoman muistin kokonaismäärät. (C.Weir; J.Noble, 2000, Memory Allocation, s26-28)



KUVA 6: Muistinvaraus muistialtaasta

3.3 Desimaalilukujen käsittely

Kaikissa mikroprosessoreissa ei ole laitteistototeutusta liukulukulaskennalle. Näissä tapauksissa voidaan käyttää kääntäjään ohjelmallisesti toteutettua emulaatiota liukulukulaskennasta. Ohjelmallisesti toteutetut liukulukulaskut vievät kuitenkin prosessoriaikaa suunnattoman paljon. Tämä ei ole useimmissa tapauksissa kovin merkittävää jos käsiteltäviä lukuja ei ole paljon ja prosessorissa on laskutehoa. Kaivusyvyysjärjestelmän päätehtävä on kuitenkin laskea mittausinformaatiota reaaliajassa. Tästä syystä ei voida käyttää kääntäjän tarjoamaa liukulukulaskentaan.

3.3.1 Liukuluvut

Liukuluvut ovat standardoitu tapa esittää desimaalilukuja 2-potenssiesityksenä. luku ilmoitetaan mantissan sekä jonkin 2 kokonaislukupotenssin tulona. Kaavassa 1 näemme liukuluku muodon matemaattisen esitystavan. Kaavassa *s* kuvaa luvun etumerkkiä, *e* eksponenttia, *m* mantissaa ja *k* kuvaa luvun kantaa. Luvun kanta on binäärijärjestelmässä 2.

$$x = (-1)^s m k^e \qquad (1)$$

Liukulukujärjestelmän vahvuudet ovat kohtalainen tarkkuus sekä laaja lukualue. Heikkouksina on kuitenkin laskennan monimutkaisuus sekä hitaus. Perus yhteenlasku toimii seuraavasti: skaalataan operandit samoille eksponenteille, summataan mantissat ja lopuksi normalisoidaan tulos sekä pyöristetään sopimaan lukualueeseen. Normalisoinnissa muutetaan mantissa siten että se on desimaaliluku jonka kokonaislukuosa on aina 1. Tätä mantissan kokonaislukuosaa ei tallenneta. Pelkästään mantissan desimaaliosa tallennetaan. Tätä mantissan, ei tallennettavaa arvoa, kutsutaan piilobitiksi. (Dr. C. Vickery, IEEE-754 Reference Material)

Taulukossa 1 on esitetty miltä liukulukuesitys näyttää binäärimuodossa. IEEE-754 – liukuluku-standardin mukaisessa perustarkkuus luvussa on varattu 8 bittiä eksponentille, 23 bittiä mantissalle sekä yksi bitti etumerkille. Etumerkin ollessa 1 luku on negatiivinen.

Etumerkki	Eksponentti	Mantissa
0	01111100	01000000000000000000000
1-bitti	8-bittiä	23-bittiä

TAULUKKO 1: IEEE -754 perustarkkuus liukuluku

Kaavassa 2 ja 3 esitetään miten taulukossa 1 esitetty liukuluvun esitysmuoto voidaan muuttaa desimaaliluvuksi.

$$(-1)^0 * 1.0100 \dots 00_b * 2^{01111100_b - 11111111_b} = x \qquad (2)$$

$$1.25 * 2^{124-127} = 0.15625 \qquad (3)$$

C/C++ kielissä on olemassa 3 liukulukutyyppiä joita kutsutaan nimillä *float*, *double* ja *long double*. Nämä vastaavat IEEE-754 standardin perustarkkuutta, kaksoistarkkuutta sekä laajennettua kaksoistarkkuutta. Perustarkkuus takaa 7 ja kaksoistarkkuus 16 merkitsevän numeron tarkkuuden. Varsinkaan kaksoistarkkuutta ei kuitenkaan pitäisi käyttää ilman laitteistopohjaista liukulukulaskentaa. Sillä liukulukujen käsittely ohjelmallisesti on hidasta. Pahimmat tarkkuusvirheet liukulukulaskennassa johtuvat pyöristysvirheistä, koska mantissa sisältää aina rajallisen määrän bittejä. Liukuluvut ovat tehokkuuspuutteistaan huolimatta silloin tällöin käyttökelpoisia, jos laskuoperaatioita on vähän ja vaaditaan laajaa lukualuetta.(D. Goldberg, 1991, http://docs.oracle.com/cd/E19422-01/819-3693/ncg_goldberg.html)

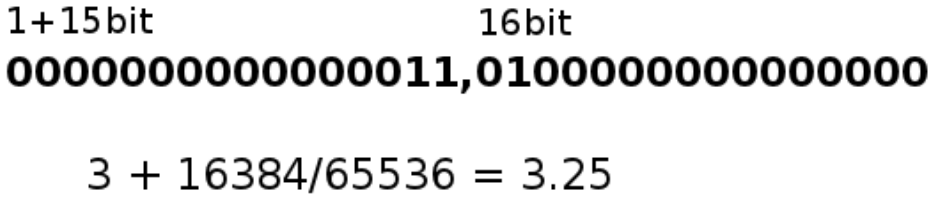
3.3.2 Kiinteän pilkun lukujärjestelmä

Järjestelmässä jossa ei ole laitteistopohjaista liukulukulaskentaa on usein järkevämpää hyödyntää kiinteän pilkun lukujärjestelmää eli niin sanottua *fixed-point* laskentaa. Normaali kokonaislukulaskenta on todellisuudessa *fixed-point* järjestelmä. Siinä on desi-

maalierotin siirretty vähiten merkitsevän bitin (LSB) oikealle puolelle. Kuitenkin normaalisti kun puhutaan *fixed-point* laskennasta, tarkoitetaan sitä, että desimaalierotin on siirretty joko eniten merkitsevän bitin (MSB) vasemmalle puolelle tai johonkin valittuun kohtaan MSB:n ja LSB:n väliltä. Jos desimaalierotin on siirretty MSB:n vasemmalle puolelle, on lukualue tällöin [-1,1] tai [0,1] mikäli ei ole käytössä etumerkkiä. (J.Fisher,P.Faraboschi,C.Young, 2005, Embedded Computing, 459)

Fixed-point lukuformaattia voidaan kuvata Q formaatilla. Esitystavasta on olemassa kaksi versiota Q*f* sekä Q*m.f* joista jälkimmäinen on kuvaavampi koska se kertoo myös käytettävän tietotyypin pituuden. Esimerkiksi Q16.16 tarkoittaa lukua jossa on 16 bittiä varattuna luvun murto-osalle ja riippuen siitä onko luku etumerkillinen vai etumerkitön siinä on 15 tai 16 bittiä kokonaisalle. Luvun mahdollinen etumerkki on aina ensimmäinen bitti. Jos käytetään esitystapaa Q16, voisi lukujärjestelmä olla joko Q0.16, Q16.16 tai Q48.16 riippuen käytettävästä tietotyypistä, joten sen käyttäminen on harhaanjohtavaa. Tässä opinnäytetyössä keskitytään lähinnä Q16.16 fixed-point lukuformaattiin.

Kokonaislukujen ensimmäinen bitti on normaalisti varattu mahdolliselle etumerkille. Etumerkittömissä luvuissa tämä ensimmäinen bitti on kokonaisosan käytössä ja kokonaisosan lukualue on tällöin suurempi. Q16.16 lukuformaatti noudattaa täysin samaa mallia normaalien kokonaislukujen kanssa. Lukuformaatissa on vain vähennetty kokonaisosasta 16 bittiä perästä ja varattu tämä tila murto-osille. Kuvassa 7 näkyy Q16.16 luvun rakenne.



KUVA 7: Q16.16 lukuformaatti

Murto-osat ovat luvun 1,0 osia. Esimerkiksi luku 0,5 murto-osa olisi heksadesimaaleina 0x8000. Kaavassa 4 näytetään kuinka murto-osasta 0x8000 saadaan tulos 0,5 jakamalla se lukuformaatin Q16.16 arvolla 1,0 joka on 0x10000.

$$\frac{fraction}{1} = \frac{0x8000}{0x10000} = 0.5 \quad (4)$$

Liitteessä 1 näytetään yksinkertaistetusti *fixed-point* laskennan perusoperaatiot. Nämä ovat kuitenkin yksinkertaistettuja esimerkkejä eivätkä sisällä esimerkiksi tarkistuksia lukualueiden ylityksiin liittyen. Esimerkistä kuitenkin voi kuitenkin huomata *fixed-point* laskennan perusasiat. Yhteen ja vähennyslasku operaatiot voidaan suorittaa suoraan käyttäen prosessorin kokonaislukulasku operaatioita sekä kerto- ja jakolaskukin onnistuu muutamalla yksinkertaisella toimenpiteellä. Varsinkin silloin kun prosessorissa on laittestotuki kerto- ja jakolaskuoperaatiolle on *fixed-point* laskenta hyvin tehokasta.

Fixed-point luvun murto-osan tarkkuuden voimme laskea käyttämällä kaavaa 5. Kaavassa Q tarkoittaa luvun murto-osan bittien määrää.

$$Q\ \frac{log(2)}{\log(10)} = d \quad (5)$$

Fixed-point lukujärjestelmän huonoja puolia verrattuna liukulukuihin on se, että sillä ei päästä samaan tarkkuuteen eikä yhtä laajaan lukualueeseen. Taulukossa 2 on lueteltu lukujärjestelmien keskenäisiä eroavaisuuksia. Tarkkuudella tarkoitetaan *fixed-point* luvuissa murto-osan merkkien määrää ja liukuluvun tapauksessa koko luvun merkkien määrää.

TAULUKKO 2: Lukujärjestelmien keskenäiset erot

Tietotyyppi	Minimiarvo	Maksimiarvo	Tarkkuus	Vaadittava tila
float	$3.4x10^{-38}$	$3.4x10^{+38}$	~7	4 tavua
Q1.15	-1.0	1.0	~4.515	2 tavua
Q16.16	-32768.	32768.99984	~5.816	4 tavua
Q32.32	-2147483648	2147483648.999...	~9.633	8 tavua

Kaivusyvyysjärjestelmässä pitää ottaa huomioon puomiston kokonaispituus, joka voi olla kymmeniä metrejä. Lisäksi kaivusyvyydet voivat olla merenpintaan nähden satoja metrejä. Lisäksi laskettavien komponenttien mitat on syötettävä vähintään millimetrin tarkkuudella, jotta voidaan eliminoida pyöristysvirheet. Millimetriä tarkempia mittoja kaivinkoneen puomistosta ei mittausteknisistä syistä edes pystyisi ottamaan. Q16.16 fixed-point lukujärjestelmällä päästään riittävään tarkkuuteen sekä katetaan riittävän laaja lukualue kaivusyvyysjärjestelmän laskennassa. Q32.32 Lukujärjestelmän käyttö ei sulautetussa järjestelmässä olisi edes kannattavaa, sillä lukujen 64 bittisten lukujen käsittely olisi lähes yhtä hidasta kuin liukulukujen. Tässä opinnäytetyössä ei kuitenkaan keskitytä kaivusyvyysjärjestelmän varsinaiseen laskentaan tai laskennan virhetarkasteluun, johtuen aihepiirin luottamuksellisuudesta.

3.3.3 Trigonometriset funktiot

Kaivusyvyysmittarin päätoiminto on laskea korkeus, syvyys ja etäisyys eroja ja näyttää informaatiota laitteen käyttäjälle. Kaivinkoneen puomiston liikkeiden laskentaan tarvitaan joukko trigonometrisiä funktioita. Tässä opinnäytetyössä ei keskitytä siihen kuinka puomiston vektorilaskenta toimii, mutta käydään lyhyesti läpi kuinka fixed-point laskentaan voidaan toteuttaa yksinkertaiset matemaattiset operaatiot.

Koska järjestelmässä ei käytetä liukulukulaskentaa, ei voida käyttää C/C++ kielen tarjoamia matemaattisia operaatiota trigonometrisille funktioille. Niinpä fixed-point laskentaan lisätään toteutus taulukossa 3 esiintyville matemaattisille operaatioille. Matemaattisten operaatioiden tulee vastata käytökseltään kääntäjän cmath kirjaston vastaavia liukulukutoteutuksia, jotta ohjelmakoodi on tarpeen vaatiessa siirrettävissä käyttämään liukulukulaskentaa.

TAULUKKO 3: Hyödyllisiä matemaattisia funktioita

cos(x)	Laskee kulman kosinin
sin(x)	Laskee kulman sinin
tan(x)	Laskee kulman tangentin
acos(x)	Laskee kulman arkuskosinin
asin(x)	Laskee kulman arkussinin
atan2(x,y)	Laskee kulman arkustangentin
sqrt(x)	Laskee luvun neliöjuuren

Kun tarkastellaan sini, kosini sekä tangentti funktioita, huomataan kaavan 6 yhtälöstä, että näistä kolmesta yhtälöstä tarvitsee toteuttaa vain kaksi. Tangentti funktio voidaan laskea jos tiedetään kulman sini ja kosini.

$$tan\,\alpha = \frac{sin\alpha}{cos\alpha} \tag{6}$$

Arkusfunktioista ei tarvitse toteuttaa muuta kuin arkustangentti sillä arkuskosini sekä arkussini voidaan johtaa arkustangentista. Kaavoista 7, 8 ja 9 nähdään kuinka arkusfunktiot voidaan johtaa arkustangentista.

$$\text{asin}\,x = \text{atan}\,\frac{x}{\sqrt{1-x^2}}\,, -1 < x < 1 \tag{7}$$

$$\text{acos}\,x = atan\frac{\sqrt{1-x^2}}{x}\,, 0 < x \leq 1 \tag{8}$$

$$\text{acos}\,x = \pi + atan\frac{\sqrt{1-x^2}}{x}\,, -1 \leq x < 0 \tag{9}$$

Yhtälöistä sin, cos, atan sekä sqrt toteutetaan likiarvon laskeva toteutus. C/C++ matematiikka kirjasto sisältää myös muita funktiota, mutta kaivusyvyysmittarin toteutuksessa ei aivan kaikkia funktioita tarvitse toteuttaa.

3.3.4 CORDIC algoritmi

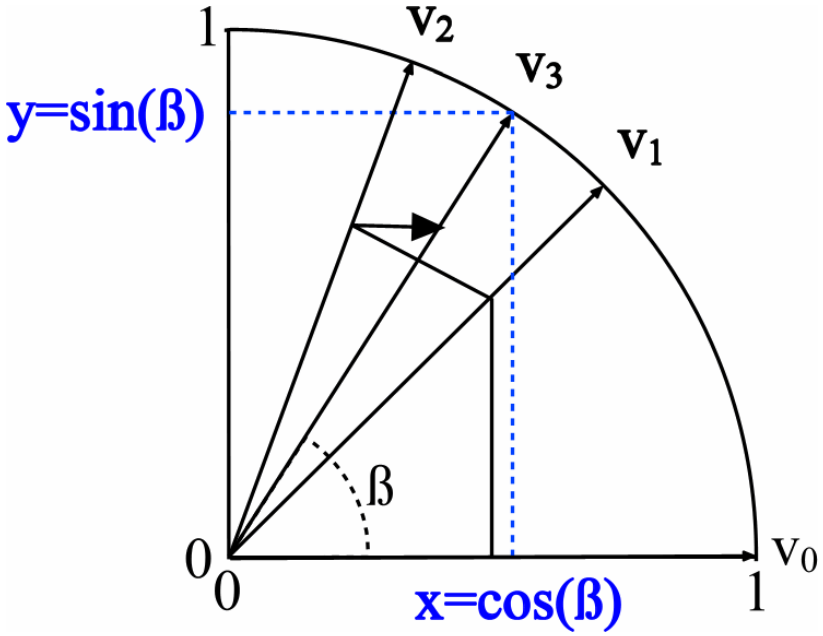
Silloin kun trigonometrisille funktioille ei löydy nopeaa laitteistototeutusta, joudutaan pohtimaan keinoja jolla pystytään laskemaan ne riittävällä tarkkuudella ja nopeudella. Kaikista yksinkertaisin sekä nopein keino on muodostaa ratkaisuista suuri etsintätaulukko, josta vain haetaan haettuun parametriin oikea vastaus. Tämä lähestymistapa kuitenkin vaatii suuren määrän ohjelmamuistia taulukon tallennukseen. Mitä tarkempia vastauksia halutaan sitä suurempi etsintätaulukko. Esimerkiksi jos luodaan sini funktiolle Q16.16 lukujärjestelmässä 4kt kokoinen etsintätaulukko päästään noin 0.088 asteen tarkkuuteen. Toinen tapa laskea sinin likiarvo on laskea se Taylorin sarjojen avulla. Kaavassa 10 on esitetty sini funktion Taylorin sarja. Yhtälöstä voidaan kuitenkin huomata, että se vaatii paljon prosessorin laskennalle hankalia jakolaskuoperaatiota sekä potenssiin korotuksia. Lisäksi Taylorin sarjasta pitäisi laskea hyvin monta derivaatan kohtaa, jotta tulos olisi tarkka.

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \tag{10}$$

Ongelman ratkaisuun on kuitenkin kehitetty paremmin digitaalisiin järjestelmiin sopivia ratkaisuja. Jack Volder kehitti vuonna 1959 tekniikan jolla voi laskea yleisimmät trigonometriset funktiot tehokkaasti käyttäen vain lyhyttä etsintätaulukkoa sekä käyttäen vain bittisiirto ja yhteenlasku operaatioita. Algoritmiä kutsutaan yleisesti CORDIC algoritmiksi, joka tulee sanoista Coordinate Rotation Digital Computer. CORDIC algoritmin vahvuuksia on lisäksi se että algoritmin jokainen iteraatio kasvattaa tarkkuutta yhdellä bitillä. Algoritmi soveltuu hyvin reaaliaikajärjestelmiin sen determistisyyden vuoksi (J.Volder,1959,the cordic trigonometric computing technique). CORDIC algoritmiä on käytetty muun muassa CORDIC I ja CORDIC III navigointi tietokoneissa, HP:n valmistamissa laskimissa sekä alkuperäisessä Intel 80x87 liukuluku laskentayksikössä.

Algoritmi perustuu havaintoon siitä, että jos pyöräytetään yksikkövektoria kulmassa z sen uusi päätepiste x,y on (cos z, sin z). Algoritmissä pyöräytetään vektoria taulukossa olevalla esilasketulla kulmalla suuntaan joka riippuu siitä onko nykyinen vektorin kulma pienempi vai suurempi kuin alkuperäinen kulma. Kun taulukko on iteroitu läpi ja ollaan päästy lähelle alkuperäistä kulmaa, yksikkövektorin y koordinaatti on kulman

sinin likiarvo ja x kulman kosinin likiarvo (Ken Turkowski, 1990, Fixed-Point Trigonometry with CORDIC Iterations). Kuvassa 8 näkyy CORDIC algoritmin toiminta yksinkertaistetusti. Vektorin pyöräytyskulmat suppenevat iteroinnin edetessä.



KUVA 8: CORDIC algoritmin toimintaperiaate
(http://en.wikipedia.org/wiki/File:CORDIC-illustration.png)

CORDIC algoritmin vahvuus piilee siinä, että kun valitaan esilaskettuun taulukkoon sopivia arvoja, jotka noudattavat kaavaa 11 voidaan x ja y pisteet laskea yksinkertaisesti yhteenlaskuilla ja vähennyslaskuilla sekä bittisiirroilla. Liitteessä 2 näkyy Q16.16 fixed-point lukuformaatin toteutus CORDIC algoritmistä.

$$\alpha_i = \operatorname{atan}\ 2^{-i} \qquad (11)$$

3.3.5 Newton-Rhapson

Neliöjuuren voi myös laskea käyttäen CORDIC algorimiä, mutta neliöjuuren laskeminen on kuitenkin helpompaa ja nopeampaa käyttäen Newton-Rhapson menetelmää. Newton-Rhapsonin etuina on se, että jokainen iterointikierros kasvattaa tarkkuutta keskimäärin kahdella numerolla, mikäli siemenarvo eli niin sanottu arvaus on ollut hyvä.

Kaavassa 12 näkyy Newton-Rhapsonin metodi. Kaavaa siis iteroidaan niin kauan, että päästään haluttuun tarkkuuteen. Arvo X_0 on niin sanottu alkuarvaus. (Dan Sloughter, 2000, Newton's Method)

$$X_{n+1} = X_n - \frac{f(X_n)}{f'(X_n)} \tag{12}$$

Neliöjuuren voi laskea Newton-Rhapson metodilla muodostamalla funktion f(x) ja sen derivaatan f'(x), sijoittamalla funktiot yhtälöön ja antamalla alkuarvauksen X_0. Vaadittavien iterointien määrä riippuu siitä kuinka hyvä alkuarvaus on valittu. Liitteessä 3 on toteutus sqrt() funktiolle käyttäen Newton-Rhapson metodia.

3.3.6 Fixed-point luokka

Fixed-point lukujärjestelmää voidaan käyttää perinteisesti käyttäen C-kielisiä funktiokutsuja. Tämä tuottaa kuitenkin paikoitellen hyvin sekavaa ohjelmakoodia. Ohessa on C kielinen listaus jossa lasketaan pisteiden P_1 ja P_2 välinen etäisyys.

```
typedef struct point {
        int32_t x;
        int32_t y;
        int32_t z;

};


int foo(point P1, point P2)
{
        int length;

        int32_t tmp1 = mul(sub(P1.x,P2.x),sub(P1.x,P2.x));
        int32_t tmp2 = mul(sub(P1.y,P2.y),sub(P1.y,P2.y));
        int32_t tmp3 = mul(sub(P1.z,P2.z),sub(P1.z,P2.z));

        length = fixsqrt(add(add(tmp1,tmp2),tmp3));

        return length;
}
```

LISTAUS 2: Kahden pisteen välisen etäisyyden laskenta

Esimerkistä voidaan huomata, että monimutkaiset matemaattiset lausekkeet menevät lähestulkoon mahdottomaksi lukea, kun ne toteutetaan c-kielisillä funktiokutsuilla. Lu-

ettavuutta voidaan korjata pilkkomalla lausekkeita paloihin ja kommentoimalla. Usein kuitenkin kun päästään tiettyyn pisteeseen eivät nämäkään keinot auta.

Ongelmasta päästään eroon käyttämällä apuna C++ ominaisuuksia. Fixed-point luvusta voidaan tehdä oma tietotyyppi luomalla siitä oma C++ luokka. Kun fixed-point luokassa kuormitetaan kaikki tarvittavat matemaattiset operaattorit, voidaan luokasta muodostettuja olioita käyttää aivan kuin normaaleja C/C++ tietotyyppejä int tai float. Operaattoreiden kuormituksia tehdessä pitää olla kuitenkin huolellinen, sillä ne voivat johtaa myös ei haluttuun käytökseen.

Listauksessa 3 esitetään sama kahden pisteen välisen etäisyyden laskenta käyttäen oliopohjaista ratkaisua fixed-point laskennasta.

```
class fix16_t {
private:
        int32_t m_value;

public:
        fix16_t(int32_t x){m_value = x << 16; }
};

typedef struct point {
        fix16_t x;
        fix16_t y;
        fix16_t z;

};


fix16_t foo(point P1, point P2)
{
        fix16_t tmp_x = (P1.x - P2.x) * (P1.x - P2.x);
        fix16_t tmp_y = (P1.y - P2.y) * (P1.y - P2.y);
        fix16_t tmp_z = (P1.z - P2.z) * (P1.z - P2.z);

        return sqrt(tmp_x + tmp_y + tmp_z);
}
```

LISTAUS 3: Kahden pisteen välisen etäisyyden laskenta käyttäen oliota

4 Kaivusyvyysmittarin käyttöliittymän toteutus

4.1 Käännösympäristön optimoinnit

Edellisessä kappaleessa käytyjen tutkimusten perusteella, päätettiin jättää C++ kielen ominaisuuksista keskeytykset, ajonaikainen tyyppi-informaatio sekä STL-kirjasto. Kyseiset ominaisuudet eivät ole reaaliaikajärjestelmässä kovin tarpeellisia ja niiden aiheuttamat haitat, ovat suuremmat kuin saadut hyödyt. Kääntäjälle voidaan määrittää liput *-fno-exceptions* sekä *-no-rtti*, joilla saadaan keskeytykset ja ajonaikainen tyyppinformaatio pois käytöstä. C++-kielen ominaisuutta luokkien periytymistä haluttiin kuitenkin käyttää. Yhden puhtaasti virtuaalisen kantaluokan luonti aiheutti kuitenkin yli 100kt ylimääräisen muistijalanjäljen ohjelmamuistissa. Ylimääräisen muistijalanjäljen aiheutti kääntäjän toteutus funktiotyngästä jota kutsutaan silloin kun puhtaasti virtuaalista metodia kutsutaan. Kyseistä funktiota ei normaalitilanteessa kutsuta ikinä, sillä puhtaasti virtuaalisen metodin suoritus on kiellettyä. Kääntäjän toteutuksessa käytettiin keskeytyksiä, dynaamista muistinhallintaa, sekä STL-kirjaston syöteolioita, joiden käyttäminen vaati suurien apukirjastojen sisällyttämistä suoritettavaan ohjelmakoodiin. GCC toteutuksen tarkoitus oli tulostaa virheilmoitus ulostulo virtaan, sekä aiheuttaa keskeytys, joka päättää ohjelman suorituksen. Tämä on ihan hyvä ratkaisu normaalissa työpöytäsovelluksessa, mutta ei ole tarpeellinen sulautetussa järjestelmässä. Alkuperäinen toteutus pystytään kuitenkin kiertämään ylikirjoittamalla kyseinen funktiotynkä. Listauksessa 4 on esiteltynä tyhjä toteutus puhtaasti virtuaalisen metodin funktiokutsun tyngästä. Funktiokutsun ei tarvitse sisältää mitään, koska sitä ei pitäisi pystyä suorittamaan.

```
extern "C" void __cxa_pure_virtual()
{
        // Do nothing or print an error message.
}
```

LISTAUS 4: Puhtaasti virtuaalisten metodien funktiokutsun tynkä

Näiden optimointien lisäksi kääntäjälle tarvitsee lisäsi kertoa *-mthumb -march=armv7 -mfix-cortex-m3-ldrd* ,joilla määritellään kääntäjälle käyttöön thumb-2 käskykanta, oikea tavujärjestys, sekä oikea prosessoriarkkitehtuuri. Kääntäjäparametrien lisäksi käännösympäristö vaatii spesifisen kääntäjän linkitystiedoston, jossa asetetaan muun muassa käytetyt muistialueet.

4.2 Kantaluokat

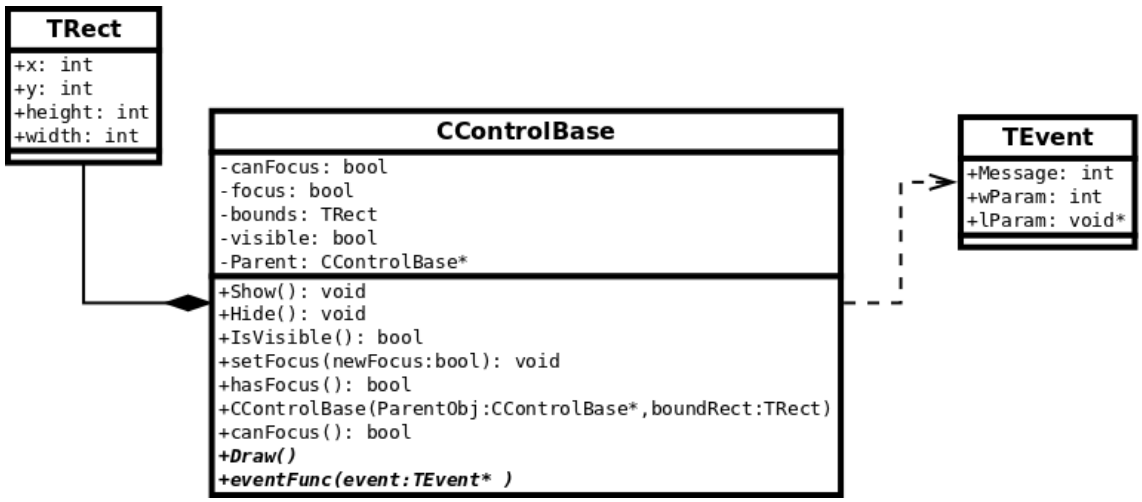
Oliopohjainen lähestymistapa on omiaan käyttöliittymien luomisessa. Yleisesti kaikilla käyttöliittymäkomponenteilla on yhteneviä ominaisuuksia. Esimerkiksi komponentilla on usein tieto sen koosta, näkyvyydestä ja sijainnista ruudulla. Lisäksi mikäli komponentin sisään on mahdollista esimerkiksi kirjoittaa, täytyy komponentilla olla tieto siitä onko se fokusoitu kyseisellä hetkellä vai ei. Kaikille komponenteille yleiset ominaisuudet voidaankin listata ja luoda näistä ominaisuuksista yksi yleinen kantaluokka, joka sisältää nämä ominaisuudet. Pohjimmaista kantaluokkaa kutsutaan tässä käyttöliittymässä kontrolliksi. Kontrolleista voidaan periytymistä hyväksikäyttäen luoda erikoistuneita luokkia kullekin käyttötarkoitukselle.

Käyttöliittymiä katsellessa voidaan usein myös huomata, että jotkut elementit menevät toisten elementtien sisälle. Näitä voidaan kuvailla siten, että suurempi elementti jonka sisälle pienemmät elementit piirtyvät, ovat suuremman elementin lapsia. Tätä sanotaan ohjelmoinnissa yleisesti emo-lapsi sukulaisuussuhteeksi. Kuvassa 9 näkyy miltä graafinen ikkuna voisi näyttää työpöytäympäristössä. Kuvasta voi havaita, että *GroupBox* nimisen elementin sisällä on kaksi lapsielementtiä ja *GroupBox* on itse pääikkunan *MainWindow* lapsi.

KUVA 9: Graafinen ikkuna PC-ympäristössä

4.2.1 Kontrollit

Kontrollit ovat tässä käyttöliittymässä kevyitä käyttöliittymäkomponentteja. Ne sisältävät perusominaisuudet liittyen niille määriteltyyn piirtoalueeseen sekä fokukseen. Kontrollit omaavat myös tiedon omasta emoluokastaan, mutta kontrollit eivät voi omistaa lapsia. Sillä kontrolli luokkaan ei ole toteutettu listaa lapsielementeistä. Koska kaikki elementit eivät sisällä lapsia, tämä on toimiva tilaa säästävä ratkaisu. Kuvassa 10 on esitetty luokkakaavio *CControlBase* luokan rakenteesta. Luokka on itsessään abstrakti kantaluokka, eikä luokasta pelkältään voida luoda olioita. Kantaluokkaa käytetään apuna erikoistuneita käyttöliittymäluokkia toteuttaessa periyttämällä kontrollin ominaisuudet. Luokkakaaviosta voidaan nähdä tummemmalla puhtaasti virtuaaliset metodit *Draw* sekä *eventFunc*. Nämä metodit tulee toteuttaa perityssä luokassa. *Draw* metodiin toteutetaan elementin piirtotapahtuma ja *eventFunc* metodiin näppäinviestien käsittelyt.



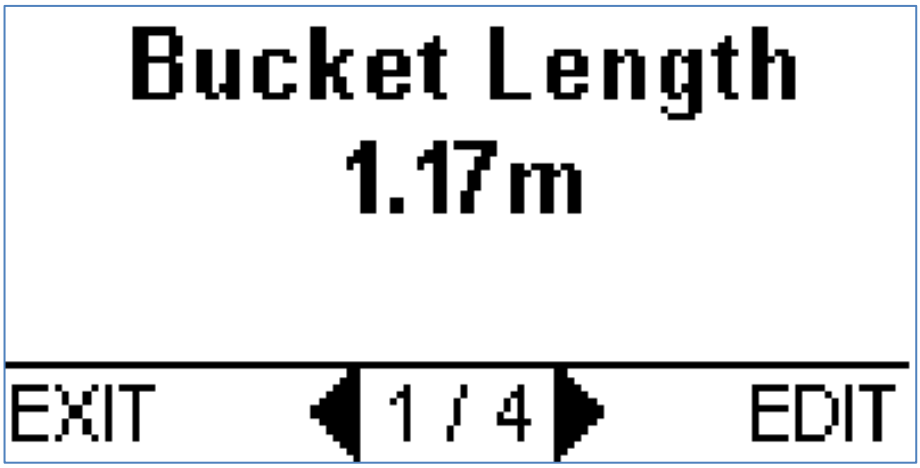
KUVA 10: CControlBase luokka

4.2.2 Ikkunat

Kohdelaitteessa on yksivärinen LCD näyttö jonka resoluutio on 128x64. Tämä asettaa suuria rajoitteita käyttöliittymän komponenttien sijoittelussa. Tästä syystä käyttöliittymässä on päätetty käyttää suurimmaksi osaksi koko ruudun kokoisia ikkunoita jotka toteuttavat yhden toiminnon.

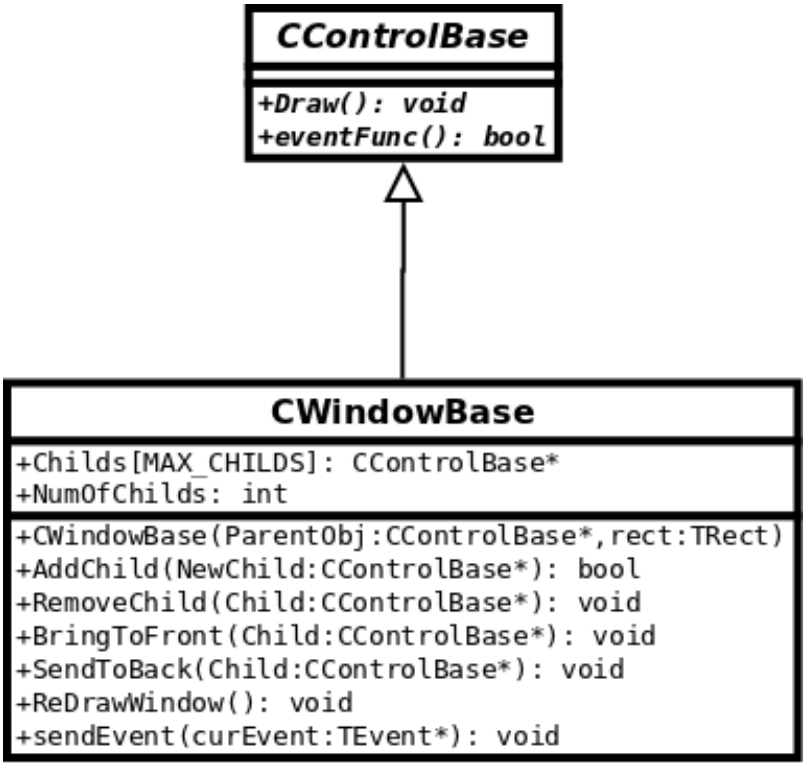
Ikkunat voivat omistaa kontrolleista poiketen lapsielementtejä. Lapsielementit voivat olla pienempiä graafisia komponentteja eli kontrolleja tai toisia ikkunoita. Kuvassa 11 näemme esimerkin ikkunaelementistä. Ikkunaelementissä on asetus velho jossa pystyy

liikkumaan sivuille, muokkaamaan arvoa laukaisemalla numeroeditorin tai poistumaan aiempaan valikkoon.



KUVA 11: WizardWindow

Ikkuna sisältää kaikki samat ominaisuudet kuin kontrolli, ja tämän lisäksi joukon lisäominaisuuksia. Kuvassa 12 näemme ikkunaluokan luokkakaavion. Ikkunaluokka *CwindowBase* perii kantaluokan *CControlBase.*

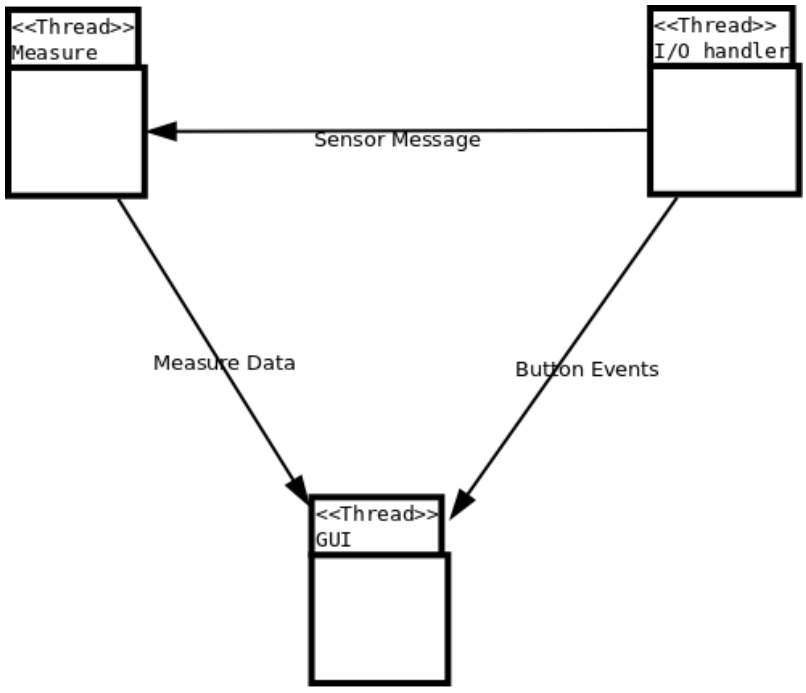


KUVA 12: CWindowBase luokka

Ikkunaluokan lisäominaisuudet kontrolliin nähden ovat lista lapsielementeistä, sekä viestien ja piirtojärjestyksen edelleen lähetys lapsielementeille. Lapsikomponentteja pystytään lisäämään ja poistamaan emoluokan lapsilistalta tarpeen vaatiessa, sekä valitun lapsikomponentin voi tuoda päällimmäiseksi. Piirronvälityksessä käytetään metodia *ReDrawWindow*, jonka sisällä ikkuna kutsuu ensin omaa piirtometodiaan ja sen jälkeen omien lapsiensa. Piirtometodia ei kutsuta, mikäli komponentti on merkitty piilotetuksi. Viestienvälityksessä viesti kuljetetaan lapsilistassa päällimmäiselle lapselle jolla on fokus. Ikkunaluokka on kontrolliluokan tapaan puhtaasti virtuaalinen ja siitä pitää toteuttaa oma luokka periyttämällä.

4.3 Viestien välitys

Järjestelmässä käytetään hyödyksi reaaliaikakäyttöjärjestelmän tarjoamia moniajo-ominaisuuksia. Käyttöliittymä tarvitsee toimiakseen tiedot painikkeiden liikkeistä. Painikeviestit luodaan viestinvälitys säikeessä kierto kyselemällä laitteistolta nappien tilaa ja tilan ollessa painettu lähettää viestinvälitys-säie viestit käyttöliittymän viestijonoon. Kuvassa 13 näemme ohjelmiston tärkeimmät prosessit ja niiden välisen viestiliikenteen.



KUVA 13: Säikeet ja viestinvälitys

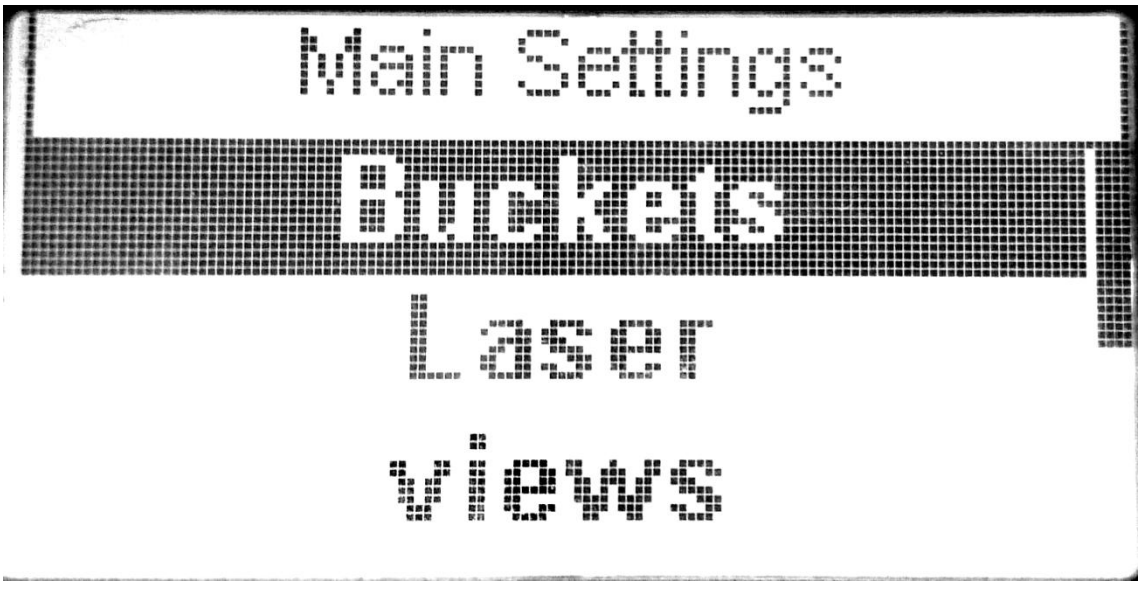
Käyttöliittymän pääsilmukassa luetaan tapahtumaviestejä viestijonosta yksi kerrallaan ja lähetetään eteenpäin käyttöliittymäkomponenteille.

4.4 Käyttöliittymäkomponentit

Käyttöliittymästä on pyritty tekemään mahdollisimman yksinkertainen ja helppokäyttöinen. Helppokäyttöisyyttä on pyritty lisäämään käyttämällä paljon yhtenäisesti toimivia käyttöliittymäkomponentteja. Tästä lähestymistavasta johtuen käyttöliittymän peruskomponentteja ei tarvitse olla järin montaa tyyppiä, vaan pyritään käyttämään samoja komponentteja eri tilanteissa. Koska järjestelmän pitää tukea vierasperäisiä merkistöjä sekä kieliä, on kaikki käyttöliittymän tekstikomponentit toteutettu tukemaan UTF-8 merkistöstandardia. Tässä luvussa esitellään lyhyesti muutama käyttöliittymäkomponentti.

4.4.1 Listavalitsin

Kaikista tärkein komponentti käyttöliittymässä on listavalitsin, jonka pohjalta kaikki valikot ja valitsimet toimivat. Listavalitsimeen voidaan liittää maksimissaan 20 mahdollista valittavaa tekstiriviä ja se palauttaa valitun tekstirivin emoluokalle signaalina. Kuvassa 14 näemme kuvankaappauksen listavalitsimesta. Listavalitsimessa haluttiin käyttää mahdollisimman isoa fonttikokoa valittaville elementeille ja tästä syystä ruudulle ei mahdu kerralla kuin 3 elementtiä sekä otsikkorivi. Listavalitsin on kuitenkin helposti skaalattavissa, jos se halutaan muuttaa isommalle näytölle sopivaksi tai käyttämään pienempää fonttia.



KUVA 14: Lista valitsin

4.4.2 Numeroeditori

Kaivusyvyysmittarissa pitää asennuksen yhteydessä pystyä syöttämään tietyt puomiston sekä koneen mitat. Tämän lisäksi käyttäjä voi haluta syöttää työmaakorkeuksia ja muita vaihtelevia arvoja mittalaitteeseen. Numeroeditorin on siis oltava selkeä ja helppokäyttöinen. Numeroeditorissa pystytään muokkaamaan kokonaislukuja, desimaalilukuja sekä murtolukuja. Kuvassa 15 on kuvankaappaus numeroeditorista.

KUVA 15: Numeroeditori

5 POHDINTA

Työn tavoitteena oli tutkia olio-ohjelmoinnin etuja sulautetun järjestelmän ohjelmistokehityksessä, sekä toteuttaa toimiva ja helposti hallittava kaivusyvyysjärjestelmän ohjelmistototeutus.

Olio-ohjelmointi mahdollistaa helpommin hallittavien ohjelmistokokonaisuuksien luomisen myös pienien resurssien järjestelmissä. Käytetty ohjelmointikieli sisältää kuitenkin paljon ominaisuuksia, joita ei pysty resurssien puutteesta johtuen hyödyntämään. Ohjelmistokehittäjällä on kuitenkin vapaus jättää nämä ominaisuudet käyttämättä ja käyttää vain niitä ominaisuuksia jotka kokee tarpeelliseksi.

Kaivusyvyysmittarin ohjelmistokehitys aloitettiin keväällä 2011 ja ensimmäinen julkaisuversio julkaistiin syksyllä 2011. Julkaisussa oli mukana vain laitteelta vaaditut perusominaisuudet ja ohjelmistokehitystä on jatkettu modulaarisesti lisäten lisäominaisuuksia. Jatkokehityssuunnitelmia ohjelmistolle on näillä näkymin vielä vuoden 2012 loppuun. Ohjelmistokehityksen edetessä on havaittu, että oliopohjainen lähestymistapa on toiminut halutulla tavalla, eli mahdollistanut helposti hallittavan sekä laajennettavan kokonaisuuden.

Työtä tehdessä opin paljon asioita liittyen sulautettujen järjestelmien ohjelmointiin. Lisäksi opin paljon asioita myös liittyen siihen kuinka käyttöjärjestelmät, käyttöliittymäkirjastot sekä ohjelmointikielet sisäisesti toimivat. Opinnäytetyö antoi siis hyvää kokemusta matalan tason ohjelmistokehityksestä, ja saatu tietämys tulee olemaan hyödyksi myös esimerkiksi työpöytäsovellusten kehityksessä.

LÄHTEET

ARM, 2012, The ARM Processor Architecture, viitattu 25.3.2012
http://www.arm.com/products/processors/technologies/instruction-set-architectures.php

Atmel, Atmega128/L Datasheet Summary, Luettu 27.3.2012
www.atmel.com/Images/2467s.pdf

Charles Weir; James Noble, 2000, Small Memory Software,
http://www.smallmemory.com/book.html

CodeSourcery, 2010, Sourcery G++ Lite: ARM EABI: Sourcery G++ Lite 2010q1-188: Getting Started, https://sourcery.mentor.com/GNUToolchain/doc11463/getting-started.pdf

Cortex-M3 Technical Reference Manual r2p0, 2008
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337g/DDI0337G_cortex_m3_r2p0_trm.pdf

Dan Sloughter, 2000, Newton's Method, http://de2de.synechism.org/c3/sec36.pdf

David Goldberg, 1991, What Every Computer Scientist Should Know About Floating-Point Arithmetic, http://docs.oracle.com/cd/E19422-01/819-3693/ncg_goldberg.html

Dr. Christopher Vickery, IEEE-754 Reference Material, Luettu 9.4.2012
http://babbage.cs.qc.cuny.edu/IEEE-754.old/References.xhtml

Embedded C++ Technical Committee, 1999, The Embedded C++ specification,
http://www.caravan.net/ec2plus/spec.html

GNU, 2011, GCC lähdekoodi, GCC-4.6-2011.09/libstdc++v3/libsupc++/eh_alloc.cc,
http://gcc.gnu.org/gcc-4.6/

Joseph A. Fisher, Paulo Faraboschi, Cliff Young, 2005, Embedded Computing

Karl Andersson, A comparison between FreeRTOS and RTLinux in embedded real-time systems ,viitattu 25.3.2012, www.iiice.net/~ice/temp/rtproj.pdf

Petteri Aimonen, 2012, Libfixmath benchmarks, viitattu 28.3.2012,
http://code.google.com/p/libfixmath/wiki/Benchmarks

Richard Barry, 2010, FreeRTOS, viitattu 25.3.2012,
http://www.freertos.org/

Shyam Sadasivan, 2006, An Introduction to the ARM Cortex-M3 Processor
http://www.arm.com/pdfs/IntroToCortex-M3.pdf

STMicroelectronics, 2011, RM0008 Reference manual,
http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/REFERENCE_MANUAL/CD00171190.pdf

STMicroelectronics, 2009, STM32F103xx datasheet

Tampereen Teknillinen Yliopisto, OHJ-4301 Sulautettu ohjelmointi. 6. Skedulointi eli vuoronnus, viitattu 25.3.2012, www.cs.tut.fi/~sulo/pruju/sulo-pruju-6.pdf

LIITTEET

Liite 1. Simplified Q16.16 math operations

```
/**
  *         Simplified Q16.16 basic math operations
  *         No overflow detection and uses 64bit integers
  *         Author: Jarno Mäkipää (2012)
  *
  */



#define HALF   (1 << (15))
#define ONE    (1 << (16))


int32_t add(int32_t a, int32_t b)
{
            return (a+b);

}

int32_t sub(int32_t a, int32_t b)
{
            return (a+b);

}

int32_t mul(int32_t a, int32_t b)
{

            int64_t temp;

            temp = (int64_t) a * (int64_t) b;  //multibly. result is Q32.32

            temp += HALF;           //round value up by adding 0.5 (0x8000)

            //convert result back to Q16.16 and return
            return (int32_t)(temp >> 16);

}

int32_t div(int32_t a, int32_t b)
{
            int64_t temp;
            // pre-multiply by the base.
            temp = (int64_t)a << 16;
            // round up
            temp = temp+b/2;


            return (int32_t)(temp/b); //divide and return

}
```

Liite 2. Simplified Q16.16 CORDIC sin cos algorithm

```
/**
  *         Simplified Q16.16 cordic sin cos algorithm
  *     Range is between -pi/2 and pi/2
  *     No overflow detection implemented
  *         Author: Jarno Mäkipää (2012)
  */

static int32_t cordic_table[16] =              /* atan(pow(2,-i));*/
{           0xC90F,  0x76B1,  0x3EB6,  0x1FD5,
            0xFFA,  0x7FF,  0x3FF,  0x1FF,
            0xFF,  0x7F,  0x3F,  0x1F,
            0xF,  0x7,  0x3,  0x1
}

/* Calculates sin and cos
 * by performing cordic rotations
 *
 * Init:
 * X = 0;
 * Y = 1 / K; //where K is constant 1.64676...
 * angle = angle;
 * Iterations:
 * Y(Yi+1)              = Yi -+ 2 ^ -i * Xi
 * X(Xi+1)              = Xi +- 2 ^ -i * Yi
 * A(Ai+1)              = Ai +- tan^-1(2^-i)
 * Result:
 * X = cos(angle)
 * Y = sin(angle)
 *
 * K = product(1 +tan^2 x^i)^(1/2)
 * K -> 1.64676... when rotations are always atan(pow(2,-i));
 */
void fixed_cordic ( int* sin, int* cos, int angle)
{
            int32_t x = 0;
            int32_t y = 0x9B75; //0.607252935 -> Q16.16
            int32_t *ctab = cordic_table;
            int16_t i = 0;

            while(i < 16) {
                        int32_t y_temp = y;
                        if(angle > 0) {
                                    y += x >> i;
                                    x -= y_temp >> i;
                                    angle -= *ctab;
                        }
                        else {
                                    y -= x >> i;
                                    x += y_temp >> i;
                                    angle += *ctab;
                        }
                        ++i;
                        ++ctab;
            }
            *sin = y;
            *cos = x;
}
```

Liite 3. Q16.16 sqrt

```
/**
  *         Simplified Q16.16 sqrt
  *         No overflow detection implemented
  *         Author: Jarno Mäkipää (2012)
  *
  */


/** calculates sqrt using newton-rhapson
  * newton-rhapson:
  *
  *  Xn+1 = Xn - f(Xn) / f'(Xn)
  *
  *  f(x)  x^2 - y = 0
  *  f'(x) 2x = 0
  */
int32_t fixsqrt(int32_t x) {

            int64_t Xn = (x >> 1) + 0xDEAD; //guess
            int64_t Xs = (int64_t)x;
            int16_t n = 6;

            while(--n){
                        Xn -= ((((Xn*Xn >> 16) - Xs) << 16) /
                                    ((0x20000*Xn)>>16));
            }

            return (uint32_t) Xn ;

}
```